Abdulla Qəhrəmanov
İlahə Cəfərova

Python

proqramlaşdırma dili



Abdulla Qəhrəmanov, İlahə Cəfərova

PYTHON

proqramlaşdırma dili

Bakı – 2015

Python dilinin populyarlığının başlıca səbəblərindən biri digər proqramlaşdırma dillərinə nisbətən daha sadə olmasıdır. Python dilinin sintaksisi çox aydın və anlaşıqlıdır.

Sizə təqdim edilən "Python proqramlaşdırma dili" adlı bu kitab Azərbaycan dilli oxucu üçün nəzərdə tutulmuşdur. Bu kitabda material sadə dildə verilmişdir və məzmun nümunələrlə müşayiət olunur.

Python dilini ilk dəfə öyrənənlər və bu dildə baza bilikləri olanlar bu kitabdan bəhrələnə bilərlər.

ISBN 978-9952-8293-1-0

Əziz oxucu!

2015-ci il Azərbaycan təhsilində çoxlu sayda novator ideyaların tətbiqi ili kimi yadda qalacaq. Bu yeniliklərdən biri də İnformatika fənninin tədrisində Python proqramlaşdırma dilinin təhsil prosesində istifadə edilməsidir. Bu o deməkdir ki, Python dilinin tətbiqi ilə Azərbaycanda proqramlaşdırma sahəsində yeni dövr başlayır. Əlbəttə ki, Azərbaycan dilində bu sahədə ədəbiyyatın olması bu işin daha da səmərəli təşkilində böyük rol oynayır. Təəssüf ki, bu günə qədər Python proqramlaşdırma dili üzrə ciddi bir ədəbiyyat mövcud deyil. 2015-ci ildə nəşr edilən VIII sinif üçün "İnformatika" dərsliyindəbu dil haqqında müəyyən həcmdə məlumat verilmişdir. Ancaq bu məlumatın azlığı Azərbaycan dilli oxucu üçün bu dilə olan marağı tam ödəmir.

Son illərdə Python dilinin populyarlığı çox artıbdır. 2015-ci ilin müxtəlif reytinq cədvəllərinə görə dünyanın ən yaxşı proqramlaşdırma dilləri arasında Python dili 3-6-cı yerlərdə yerləşir. ABŞ-da təhsil sistemində istifadə edilən proqramlaşdırma dilləri arasında Python dili birinci yer tutur. Bir çox şirkətlər öz Web-layihələrində (məsələn, Google, YouTube, Yandex və s.) Pythondan olduqca geniş istifadə edirlər. Bu sıraya MIT (Massaçusets Texnologiya İnstitutu) və NASA da daxildir.

Python dilinin populyarlığının başlıca səbəblərindən biri digər proqramlaşdırma dillərinə nisbətən daha sadə olmasıdır. Python dilinin sintaksisi çox aydın və anlaşıqlıdır.

Python dilinin populyar olmasının səbəblərindən biri də İnternetdə çoxlu sayda Python sevərlərin icmalarının olmasıdır.

Təəssüflər ki, Azərbaycanda bu günə kimi bu dil populyarlıq qazanmayıb. Düşünürəm ki, 2015-ci il Python dilinin Azərbaycanda tətbiqi üzrə dönüş ili olacaq.

Python dili ilk dəfə proqramlaşdırmanı öyrənənlər üçün ən əlverişli dildir.

Sizə təqdim edilən "Python proqramlaşdırma dili" adlı bu kitab bu boşluğu doldurmaq üçün edilən ilk cəhdlərdən biridir (bəlkə də birincidir). Bu kitabın hazırlanmasında çoxlu sayda mənbələrdən istifadə edilsə də, özəllilik və sadə dil üslubu əsas meyarlar olmuşdur. Çalışmışıq ki, material daha çox nümunələrlə müşayiət edilsin və dili sadə olsun. Düşünürük ki, 6 aylıq zəhmətimiz hədər getməyib. Amma yenə də son qiymət sən oxucunun ixtiyarındadır.

Kitab haqqında arzularınızı, istəklərinizi və xüsusilə tənqidi fikirlərinizi aşağıdakı ünvanlara bildirə bilərsiniz:

***e-mail:* abdullaqehreman@gmail.com**

***Telefon:* (+99450) 3503920**

Hörmətlə: ***Abdulla Qəhrəmanov***

İlahə Cəfərova

Mündəricat

PYTHON (Ön söz əvəzi)

Python —masaüstü kompüterlər üçün nəzərdə tutulan bütün əməliyyat sistemlərində işləyən proqramlaşdırma dilidir. Python proqramlaşdırma dili 20 ildən artıqdır ki, yaradılıb və hələ də daim təkmilləşdirilir. Python dilinin yaradılması ideyası 1980-cı illərdə yaranıb və 1989-cu ilin dekabr ayında Hollandiya (Niderland) riyaziyyat və informatika mərkəzinin işçisi Qvido van Rossum (Guido van Rossum) tərəfindən yaradılmağa başlanıb.

Qvido o dövrdə Britaniyanın BBC kanalının "Monti Python hava sirki" adlı komediya seriyasının fanatı idi. Bu proqramın şərəfinə yaratdığı dili Python adlandırıb. Yəni ilkin variantda dilin adının piton ilanı ilə heç bir əlaqəsi olmayıb. Sonradan dilin loqosunda piton ilanının şəklindən istifadə edilməyə başlanıb.

16 oktyabr 2000-cildə dilin populyar 2.0 versiyası istifadəyə verilib.

Python 3.0 versiyası 3 dekabr 2008-ci ildə işıq üzü görüb.

Hal-hazırda onun köhnə 2-ci versiyası və müasir 3-cü versiyası mövcuddur. Bu versiyalar arasında ilk baxışda o qədər də fərq görünməsə də, prinsipial uyğunsuzluqlar var. 2-ci versiya artıq inkişaf etdirilmir, ancaq hələ də işlənir. Çünki bu versiya üçün çoxlu sayda proqramlar və kitabxanalar hazırlanıb. Biz 3-cü versiyadan istifadə edəcəyik. Bu kitabda verilən bütün nümunələr 4 fevral 2015-ci il buraxılışı olan Python 3.4.3 versiyasında yoxlanılıb. Düşünürük ki, gələcək versiyalarda elə də prinsipial dəyişikliklər olmayacaq.

Python — interpretasiya edilən müasir universal proqramlaşdırma dilidir. İnterpretasiya edilən proqramlaşdırma dilinin əsas xüsusiyyəti ondan ibarətdir ki, bu dildə yazılan ilkin proqram kodu birbaşa mərkəzi prosessor ilə yerinə yetirilən (kompilə edilən dillərdən fərqli olaraq) maşın koduna çevrilmir. Xüsusi proqram-interpretator vasitəsilə icra edilir. Onun üstünlükləri aşağıdakılardır:

1. Krossplatformalı və pulsuzdur. **Krossplatformalı** dedikdə yazılmış proqram kodunun bir neçə əməliyyat sistemində və ya aparat platformasında işləyə bilmək qabiliyyəti başa düşülür.
2. Sadə sintaksisə malikdir və zəngin xüsusiyyətləri qısa zamanda və aydın formada proqram yazmağa imkan verir.
3. Sadəliyinə görə dil Basic dili ilə müqayisə edilə bilər. Ancaq bol imkanlara malikdir və daha müasirdir.
4. Proqram kodu keyfiyyətlidir. Python dilində proqram kodu asan oxunur, vahid tərtibat formasının olması onun başa düşülməsini asanlaşdırır.
5. Məhsuldarlığın yüksək olması. Python dilində proqram kodu yazmaq üçün sərf edilən vaxt digər dillərə nəzərən azdır.
6. Python dilində yazılmış proqram kompilə edilmədən, əlaqələr qurulmadan həmin andaca yerinə yetirilir.
7. Bol standart kitabxanaya malikdir, sənayeyönümlü əlavələr (şəbəkə ilə iş, GUI, verilənlər bazası və s.) yazmağa imkan verir.
8. İnteqrasiya komponentləri. Python dili digər dillərin kitabxanalarından və əksinə digər dillər Python dilinin kitabxanalarından faydalana bilərlər.

9. Python dili UNICOD kodlaşma ilə də işləyir. Bu o deməkdir ki, sırf Azərbaycan hərflərindən proqram kodunda istifadə etmək mümkündür.

Düzdür, Python dilində Azərbaycan klaviaturu ilə işləyən zaman “ə” hərfini sıxdıqda “?” işarəsi çıxır. Buna səbəb klaviaturun bir baytlıq kodlaşma (ASCII) ilə işləməsidir. “Ə” hərfinin Python dilində görünməsi üçün belə bir fənddən istifadə etmək olar. Tək “ə” hərfini və ya tərkibində “ə” hərfi olan mətn parçası ixtiyari mətn redaktorunda (məsələn, MS Word) yazılır. Sonra həmin hərf və ya mətn parçası mübadilə buferi vasitəsilə Python dilinin proqram koduna əlavə edilir. Bu proses hamıya bəllidir. Ardıcıllıq belədir:

- **Tək “ə” hərfini və ya tərkibində “ə” hərfi olan mətn parçası ixtiyari mətn redaktorunda (məsələn, MS Word) yazılır.**
- **Mətn redaktorunda yazılmış mətn parçası seçilir.**
- **Ctrl+C.**
- **kursor Python dilində yazılmış proqram kodunda həmin parça əlavə ediləsi yerə qoyulur.**
- **Ctl+V.**

Son dövrlərdə dünyada İnformatika fənni üzrə bir çox məktəb olimpiadalarında Python dilindən istifadə edilir.

Python dilinin imkanları

Pyton dilinin aşağıdakı imkanları var:

- xml/html fayllarla iş
- http sorğularla iş
- GUI (qrafik interfeys)
- Veb-ssenarilərin yaradılması
- FTP protokolla iş
- Təsvirlərlə, audio və video fayllarla iş
- Robottexnikası
- Riyazi və elmi hesablamaların proqramlaşdırılması
- və digər.

Beləliklə, Python gündəlik qarşılaşdığımız çoxlu sayda məsələlərin həlli üçün ən uyğundur. Python dili praktiki olaraq heç bir məhdudiyyətə malik deyil, buna görə də bir çox iri layihələrdə də isitfadə edilir. Məsələn, Python Google və Yandex kimi İT-nəhənglər tərəfindən isitfadə edilir. Sadəlik və universallıq Pythonu proqramlaşdırma dilləri arasında ən yaxşılarından biri edir.

Proqramçı düşüncəsi

Görəsən bir peşə sahibi kimi proqramçı necə düşünür?

Bu düşüncə tərzi özündə üç elementi saxlayır: riyazi, mühəndis və təbiət elmləri düşüncə tərzi. Riyaziyyatçı proqramçı formal dillərdən istifadə edərək öz ideyalarını (konkret olaraq alqoritmləri) ifadə edirlər. Mühəndis kimi onlar alternativləri tutuşduraraq, kompromisləri taparaq yeni məhsullar (proqramlar) yaradırlar. Elmi işçi kimi onlar çətin sistemlərin özlərini necə aparmalarını müşahidə edir, hipotezlər formalaşdırırlar və edilmiş fərziyələri yoxlayırlar.

Proqramçı üçün ən vacib bacarıq məsələni həll etmək bacarığıdır.

Məsələni həll etmək bacarığı dedikdə məsələni (problemi) formalaşdırmaq, məsələnin həlləri haqqında yaradıcı düşünmək, və məsələnin həllini aydın və dəqiq ifadə etmək başa düşülür. Beləliklə, proqramlaşdırma öyrənmə prosesi – problem həll etmək bacarıqlarının inkişafı üçün böyük imkanlar verir.

Proqramlaşdırma ilə məşğul olmaq Sizin intellektual bacarıqlarınınzın inkişafına yardımçı olacaq.

Proqram. Translyator. Python proqramlaşdırma dili

Proqram konkret icraçı üçün nəzərdə tutulmuş təlimatlar toplusudur. İcraçı deyəndə müxtəlif növ kompüterlər, avtomatlar, rəqəmsal məişət avadanlıqlar və s. nəzərdə tutulur.

Bildiyimiz kimi bütün rəqəmsal avadanlıqlar (kompüter, məişət cihazları və s.) ikilik say sistemi ilə işləyən icraçılardır. Deməli həmin icraçıların konkret iş görməsi üçün hazırlanan proqramlar (təlimatlar) da ikilik say sistemində işləyirlər. İxtiyari icraçının mikroprosessoru modelindən asılı olaraq müəyyən sayda idarəedici əmrlərə malikdir. Bütün icra olunan proqramlar həmin ikilik koda malik əmrlərdən təşkil olunublar. İlk kompüterlər üçün proqramlar ikilik kodda yazılırdı. Mən özüm də 1973-cü ildə ilk proqramımı ikilik say sistemində yazmışam (A.Q.). Əlbəttə ki, bu qaydada proqram yazmaq xüsusi peşəkar hazırlıq tələb edirdi. Buna görə proqramlaşdırma işini asanlaşdırmaq üçün proqramlaşdırma dilləri yaradılmağa başladı və bu proses hələ də davam edir.

Proqramlaşdırma dili proqram yazmaq üçün (adətən kompüter üçün) nəzərdə tutulmuş formal dildir.

Proqramlaşdırma dillərinin evolyusiyasının əsas mərhələləri aşağıdakılardır:

- **Maşın kodu;**
- **Aşağı səviyyəli proqramlaşdırma dili – Assembler;**
- **Yüksək səviyyəli dillər;**
- **Obyekt-yönümlü dillər.**

Aşağı səviyyəli dil (Assembler) prinsipcə maşın kodu şəklində verilmiş əmrlərin Müəyyən hərflərlə işarə edilməsi yolu ilə yaradılmasıdır, prinsipcə bu elə maşın kodunda yazılmış əmrlərə yaxındır. Siyahıdan göründüyü kimi, evolyusiyanın daha inkişaf etmiş səviyyəsində yüksək səviyyəli dillər və obyekt-yönümlü dillər yer tutur. Bu dillərin əsas xüsusiyyəti onların insan dilinə yaxın formal dil olmasıdır. Belə çıxır ki, yüksək səviyyəli dildə və obyekt-yönümlü dildə hazırlanmış proqram kodu icraçıda (bizim halda kompüterdə) icra olunmaq üçün maşın koduna çevrilməlidir. Bu həmin dillərin əskiklikləri hesab edilə bilər. Əlbəttə ki, bu əlavə vaxt və əlavə xərc deməkdir.

Ancaq üstünlüklər çoxdur. Birinci həmin dillərdə proqram yazmaq çox asandır. Çünki, formal dil özü insana daha anlaşıqlıdı, deməli vaxt nöqteyi-nəzərincə proqram hazırlanması müddəti azalır. İkincisi, bu dillərdə yazılan proqram kodu daşınandır. Bu o deməkdir ki, həmin dillərdə yazılmış proqramlar kiçicik dəyişiklik edilməklə (bəzən də tam olduğu kimi) müxtəlif tipli kompüterlərdə icra oluna bilər. Aşağı səviyyəli dildə yazılmış proqram yalnız konkret bir növ kompüterdə icra olunmaq üçün hazırlanır. Digər növ kompüterdə icra olunmaq üçün yenidən yazılmalıdır.

Bizim öyrənəcəyimiz Python dili də yüksək səviyyəli obyekt-yönümlü proqramlaşdırma dilləri qrupuna aiddir. Yəqin ki, Siz C++, PHP, Java kimi proqramlaşdırma dilləri ilə də tanışsınız, ya da onlar haqqında eşitmişiniz.

Sadaladığımız üstünlüklər nəticəsində artıq demək olar ki, bütün proqramlar yüksək səviyyəli və obyekt-yönümlü dillərdə yazılır. Aşağı səviyyəli dillər çox az sayda xüsusi proqramları yazmaq üçün istifadə edilir.

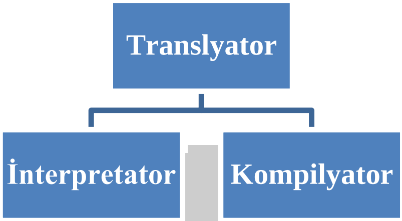
Translyator – hər hansı yüksək səviyyəli proqramlaşdırma dilində hazırlanmış proqramın ilkin kodunun maşın dilinə çevirən xüsusi proqramdır. Yəni, proqramlaşdırma dilində

yazıldıqdan sonra proqram gerçək maşın koduna çevrilməlidir ki, kompüterin mərkəzi prosessoru bunu "başa düşüb" icra edə bilsin.

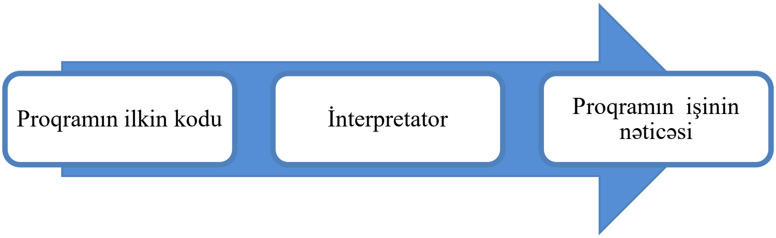
Translyatorların 2 növü var:

- **İnterpretator;**
- **Kompilyator.**

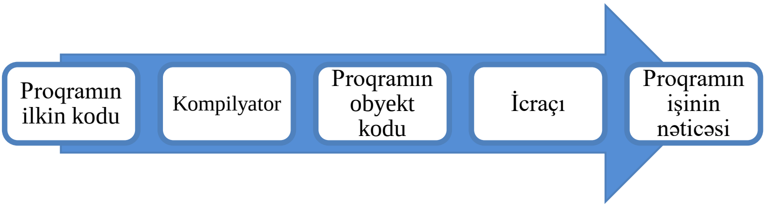
İnterpretator –yüksək proqramlaşdırma dilində yazılmış proqramı oxuyur və həmin andaca icra edir. Yəni proqramın təlimatlarına riayət edir. Proqramın ilkin kodu cümlə-cümlə oxunur və icra olunur. Bu halda o əməliyyat sistemi ilə birbaşa təmasda olur.



Siz proqramı yazırsınız, sonra başladırsınız və dərhal kompüterin hər addımda nə etdiyini görürsünüz. Əgər proqramda nəyisə dəyişmək lazımdırsa, onu yerindəcə edib proqramı yenidən translyasiya edə bilərsiniz. Ancaq bu yolun bir çatışmazlığı var: proqram tam hazır olduqdan sonra da, hər dəfə yerinə yetirilməzdən qabaq onun hər bir sətri maşın koduna çevrilir və nəticədə proqramın ümumi icra müddəti uzanır. Aşağıdakı sxemdə interpretatorun iş prisipi verilib.



Kompilyator – proqramın ilkin kodunu tam oxuduqdan sonra onu maşın koduna çevirir və icra edilən fayl yaradır. Sonradan ilkin koddan asılı olmayaraq bu fayl dəfələrlə çalışdırıla bilər. Aydındır ki, bu zaman yenidən translyasiyaya gərək qalmır.



Bir çox müasir proqramlaşdırma dilləri hər iki prosesdən istifadə edir. Əvvəlcə onlar proqramın ilkin kodunu **bayt-kod** adlanan daha aşağı dilə kompilyasiya edirlər. Bundan sonra bayt-kod **virtual maşın** adlanan proqram vasitəsilə interpretasiya edilir.

Python hər iki prosesdən istifadə edir. Düzdür, çox vaxt Python əsasən interpretator kimi istifadə olunur, ancaq bu dilin kompilyatoru da vardır.

Proqram nədir?

Yuxarıda dediyimiz kimi, Proqram konkret icraçı üçün nəzərdə tutulmuş əmrlər (təlimatlar) toplusudur. Yəni proqram hesablamanı necə aparmağı, verilənləri necə emal etməyi təsvir edir. Kompüter riyazi hesablamalar aparır, mətn, audio, video və s. formatlı verilənləri emal edir.

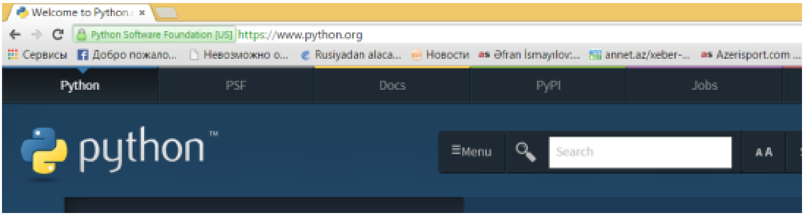
Kompüter üçün proqramlar müxtəlif proqramlaşdırma dillərində müxtəlif cür təsvir olunur. Ancaq demək olar ki, bütün dillərdə bir neçə baza konstruksiyaları mövcuddur:

GİRİŞ		
Klaviaturdan, fayldan və ya digər qurğudan verilənləri almaq (daxil etmək).		
Riyazi hesablamalar	**Şərti icra**	**Təkrar**
Riyazi hesablamalar aparır Məsələn: toplama və çıxma	**Müəyyən şərtin yerinə yetirilməsini yoxlayır. Əgər şərt ödənirsə, bu şərti ödəyən müəyyən təlimatlar ardıcıllığı yerinə yetirilir.**	**Müəyyən ardıcıllıqlı təlimatları cüzi dəyişikliklərlə dəfələrlə təkrar edir.**
ÇIXIŞ		
Verilənləri ekranda əks etdirmək, verilənləri fayla və ya digər qurğuya göndərmək.		

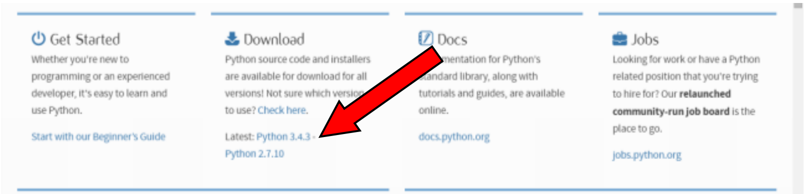
Doğrudan da əgər Siz nə vaxtsa hansısa proqramdan istifadə etmişsinizsə, çətinliyindən asılı olmayaraq, o bu konstruksiyalardan yığılıb. Beləliklə, proqramlaşdırma böyük məsələni (problemi) daha kiçik altməsələlərə bölərək baza konstruksiyalardan istifadə etməklə məsələni həll etməyə çalışmaqdır. Əgər hansısa addımda məsələni tam həll etmək olmursa, onu daha altməsələlərə bölüb prosesi davam etmək lazımdır. Bu proses altməsələlər daha sadə olana və məsələ tam həll edilənə qədər davam edir.

Python proqramını yüklənməsi və quraşdırılması

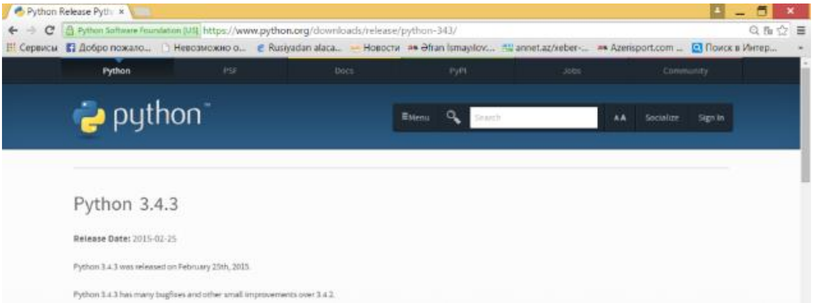
Bu vəsaitdə yalnız MS Windows ƏS üçün Python proqramının kompüterinizə quraşdırılması haqqında danışacağıq. İşə başlamamışdan əvvəlPython proqramınındistributorunu kompüterinizə yükləməlisiniz. Bunun üçün**python.org** rəsmi saytına daxil olmaq lazımdır.



Saytın aşağısında Download linki (hipermüraciəti) var. Bu vəsait yazılan anda Python 3 üçün Python 3.4.3 versiyası mövcud idi. Həmin linkə tıklayıb növbəti səhifəyə daxil oluruq.

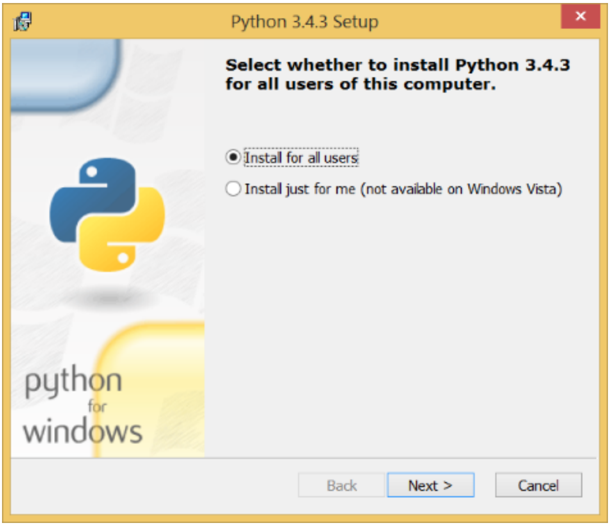


Növbəti açılan səhifədə Python 3.4.3 versiyasının müxtəlif fayllarına müraciətlər yerləşdirilib. Səhifənin aşağısında Files bölməsində Windows ƏS üçün Python proqramının linkləri var. Sizin kompüterinizin prosessorunun 32 bitlik və ya 64 bitlik olmasından asılı olaraq Windows x86 MSI installer (32 bitlik) və ya Windows x86-64 MSI installer (64 bitlik) variantını seçib tıklayın.

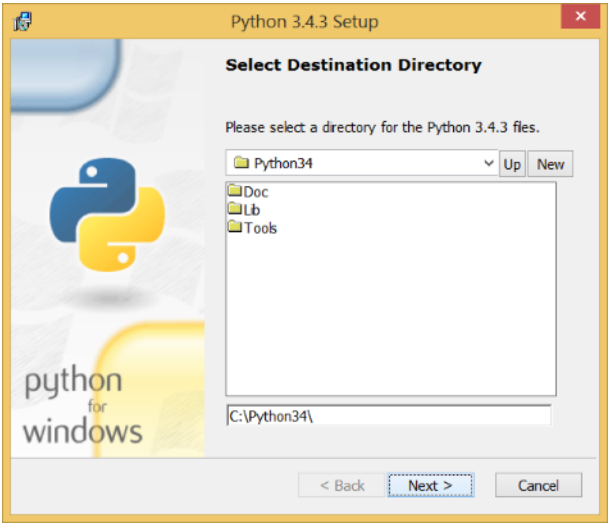




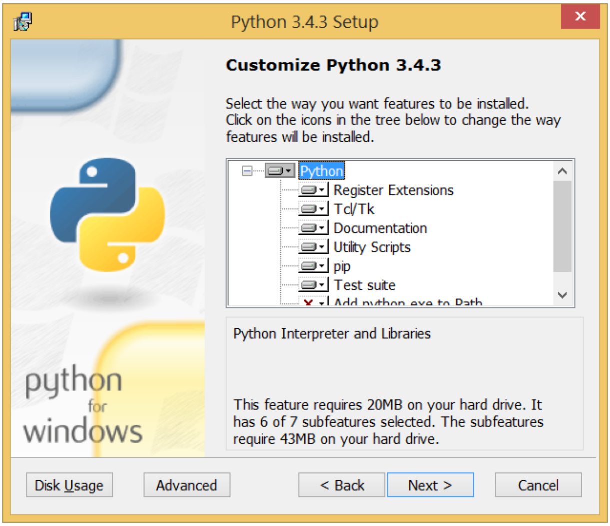
Yükləmə bitdikdən sonra **yükləmələr** (adətən **Downloads**) qovluğundan prosessorun növündən asılı olaraq yüklənmiş **python-3.4.3.msi** və ya **python-3.4.3.amd64.msi** proqramını işə salırıq. Adi proqram quraşdırma prosesi gedəcək. Növbəti menyuda iki variantdan biri seçilir (seçimi Siz edirsiniz).



Quraşdırma üçün qovluq seçilir. Siz istədiyiniz qovluğu seçə bilərsiniz. Ancaq susma halında təklif edilən qovluğu seçmək məsləhətdir.



Quraşdılacaq komponentləri seçirik. Əgər əminsinizsə, seçimi susma halı üçün seçin.



Python proqramının quraşdırılmasını gözləyirik...

Finish. Artıq Python Sizin kompüterdə quraşdırılıb! Siz ilk proqramınızı yazmağa hazırsınız.



İlk proqram. IDLE örtük mühiti

IDLE mühitində ilk proqramımızı yazaq.

Python proqramı yükəndikdən sonra ilk açılan interfeys IDLE örtük mühitidir. **IDLE** (**I**ntegrated**D**eve**L**opment**E**nvironment) — bu Python dilində inteqrasiya olunmuş proqram yaratma mühitidir. Rəsmi olaraq qısaca IDE adlanır. Vəsaitdə verilən nümunələr Windows 10ƏS-də yoxlanılıb.

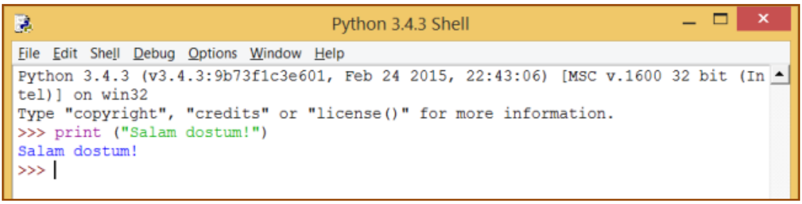
Öncə interaktiv rejimdə **IDLE Python örtüyünü**(Python Shell) çağırırıq. Sonuncu sətirdə ardıcıl gələn üç "böyükdür"(>>>) yazılıb. Bu işarə çağırış adlanır,işarə olan sətir isə əmr sətridir. Belə sətirdə istənilən proqram kodunu yazmaq olar. Birinci əmrimiz

```
print ("Salam dostum!")
```

olacaq. Əmri yazıb **Enter** klavişini sıxan kimi növbəti sətirdə

```
Salam dostum!
```

mətni görünəcək.



Təbrik etmək olar. Bu bizim Python proqramlaşdırma dilində yazdığımız ilk proqramdır. İnteraktiv rejimdə işləmə qaydası ilə tanış olandan bu rejimdə işləmək bacarığımızı təkmilləşdirək. Məsələn, aşağıdakı əmrləri yazaq

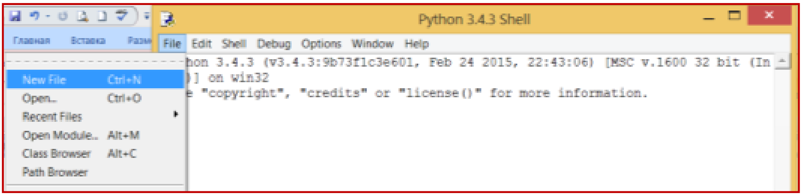
```
print(3 + 4)
print(3 * 5)
print(3 ** 2)
```

Nəticə açağıdakı kimi olacaq:

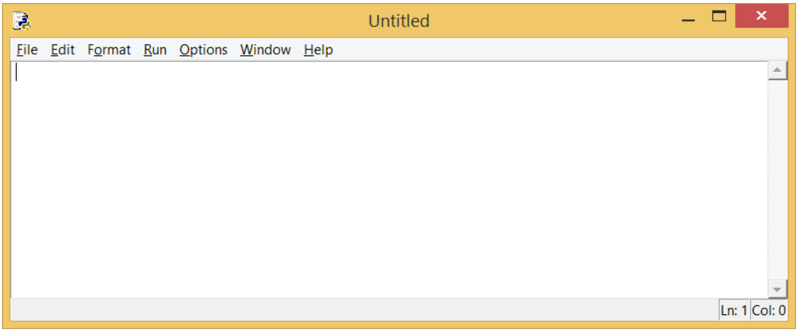
```
>>> print(3 + 4)
7
>>> print(3 * 5)
15
>>> print(3 ** 2)
9
```

Amma bizim əsas məqsədimiz interaktiv rejimdə işləmək deyil. Biz proqram kodunu fayl şəklində yadda saxlayıb proqramı oradan çağırmaq istəyirik.

Bunun üçünIDLE interaktiv rejimində ya **File → New File**seçirik, yada **Ctrl+N** klavişlərini sıxırıq.



Nəticədə mətn redaktoruna oxşar yeni pəncərə açılır. O Python redaktoru adlanır. Proqram kodu bu pəncərədə yazılacaq.



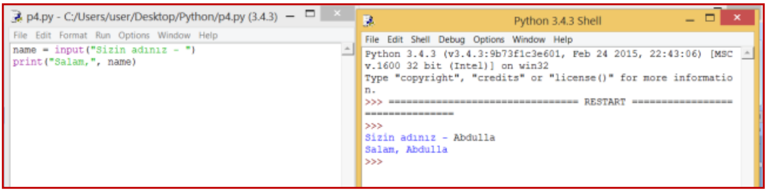
Açılmış pəncərədə növbəti əmrləri yazın:

```
name=input('Sizin adınız –')
print('Salam, ',name)
```

Birinci sətirdə 'Sizin adınız –' sorğusuna hansısa cavab verilməlidir (bu nümunədə **Abdulla**) və **Enter** klavişini sıxılmalıdır. Ad daxil edilib **Enter** klavişi sıxılmayana qədər ikinci əmr yerinə yetirilməyəcək. Bu proqram kodunun işləməsi nəticəsində name dəyişənində daxil edilmiş informasiya (sizin adınız) yadda saxlanacaq və sonra ekranda **Salam** sözündən sonra həmin ad çap ediləcək.

İndi **F5** klavişini (və ya IDLE menyusunda **Run → Run Module**əmrini seçmək də olar) sıxmaq proqramı yerinə yetirmək olar. Proqram ilk dəfə yerinə yetiriləndə proqram mətninin faylda yadda saxlanılması istəniləcək. Mətni istədiyiniz yerdə yadda saxlaya bilərsiniz. Yalnız bundan sonra proqram yerinə yetiriləcək.

Siz sonda təqribən aşağıdakı kimi nəticə alacaqsınız. Solda proqram mətni, sağda isə nəticə görünür:



Gördüyünüz kimi proqramın başlamasını Python örtüyündə

>>>==================== RESTART ==================== sətrinin yazılması ilə bilmək olar.

Gəlin aşağıda verilən bir neçə əmrin işləməsini müstəqil yoxlayaq:

```
d1=10
d2=20
d3=d1+d2
print(d3)
a=7; b=3; print(a, b)
metn="Yoxlama"
print(metn[2])
print(metn[:2])
print(metn[2:5])
print(metn[:-1])
print(len(metn))
print("123"+"456")
print("123"*3)
```

Əgər nəticələr aşağıdakı kimidirsə, deməli düzgün yoldayıq. Nəsə alınmasa, təəssüflənməyin və yazdığınız əmrlərdəki səhvi tapıb düzəliş edin və təkrar yerinə yetirin.

```
>>>
30
7 3
x
```

```
Yo
xla
Yoxlam
7
123456
123123123
```

Siz özünüz yazdığınız əmrlərin işləmə prinsipini sərbəst analiz edin. Hələ ki, yazdığımız əmrlərə şərh verməyəcəyik. Gələcəkdə ayrı-ayrı bölmələrdə oxşar əmrlərlə daha dərindən tanış olacağıq.

İkinci təbrik! Siz artıq sadə proqram yazmağı bacarırsınız və IDLE proqram hazırlama hazırlama mühiti ilə tanışsınız.

Obyekt

Obyekt Python dilində proqramlaşdırmada ən fundamental anlayışlardandır.

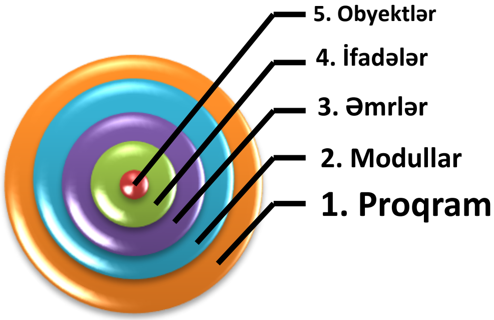
Python dilində yazılmış proqramları aşağıdakı hissələrə ayırmaq olar:

Modullar – Təlimatlar – İfadələr – Obyektlər

Daha da açıqlasaq

- **Proqram modullara bölünür;**
- **Modullar əmrlərdən (təlimatlardan) ibarətdir;**
- **Əmrlər (Təlimatlar) ifadələrdən ibarətdir;**
- **İfadələr obyektlərdən təşkil olunub və onları idarə edir..**

Bu sxemdə dediklərimiz sxematik təsvir olunub.



Qiymətlər və tiplər

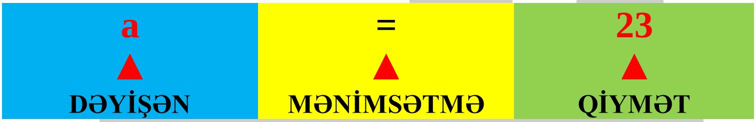
Riyaziyyatdan bizə məlum olan bəzi anlayışları yada salaq.

OPERANT1	OPERATOR	OPERANT2
4	+	**5**
a	=	**23**
a	==	**b**
b	>	**0**

İstənilən 2 operant arasında münasibət operator vasitəsilə tənzimlənir. Python dilində aşağıdakı operatorlardan istifadə edilir: **+, -, *, /, **, //, %, >, <, >=, <=, !=**. Bu və digər operatorların işləmə prinsipi haqqında sonrakı bölmələrdə izahat veriləcək.

Python dilində olan daha iki anlayış ilə tanış olaq.

Qiymət – sərbəst istifadə edilən və ya dəyişənlərə mənimsədilən müxtəlif formatlı məzmundur.



Yuxarıda göstərdiyimiz nümunələrdə biz qiymətləri görmüşdük:

```
7
Salam dostum!
```

Burada iki qiymət var. 7 – tam ədəd olan məzmundur. “Salam dostum!” isə simvollar ardıcıllığından ibarət sətir tipli məzmundur.

Yuxarıda deyildiyi kimi, qiymətlərdən ya sərbəst istifadə etmək olar, ya da dəyişənə mənimsətmək yolu ilə. Məsələn, **ad** = **“Aysu”** və ya **a** = **23**. Burada **Aysu** və **23** qiymətdirlər.

Print əmri nəinki sətirlərlə, həm də tam ədədlərlə də işləyir.

```
>>> print 4
4
>>> print (4)
4
```

Əgər Siz hər hansı qiymətin hansı tipə malik olduğuna əmin deyilsinizsə, interpetator bunu Sizə deyə bilər.

```
>>> type("Salam dostum!")
<type 'str'>
>>> type(21)
<type 'int'>
```

Sətir (ing: string) **str** tipinə aiddir, tam ədədlər isə (ing: integer) **int** tipinə.

Tam və kəsr hissəsi olan ədədin tipi **float** (ing: üzmək) olacaq. Bəzi ölkələrdə riyaziyyatda tam ədədlə kəsr hissəni ayırmaq üçün vergül işarəsindən istifadə edilir. Python dilində digər alqoritmik dillərdə olduğu kimi, ədədin tam hissəsi ilə kəsr hissəsini ayırmaq üçün nöqtədən istifadə edilir və o **onluq nöqtə** adlanır.

```
>>> type(2.3)
<type 'float'>
```

Python dilində 2,3 yazsaqtamam başqa nəticə alarıq:

```
>>> print 2,3
2 3
>>> print (2,3)
23
```

Python 2,3 yazılışını iki elementdən ibarət olan siyahı kimi interpretasiya edir. Əgər "23" və ya "2.3" yazsaq onda qiymət sətir olacaq. Çünki yazılış dırnaq arasındadır.

```
>>> type("23")
<type 'str'>
>>> type("2.3")
<type 'str'>
```

Dəyişənlər

Python proqramlaşdırma dilinin əsas güclü tərəflərindən biri **dəyişənlərlə** işləmək imkanıdır. **Dəyişən** qiymətə müraciət edənə addır.

Mənimsətmə cümləsi yeni dəyişən yaradır və ona qiymət verir. Qiymətin özü dəyişən ola bilər. Məsələn**:** **a=5; b=3;** **c = a** ;

```
>>> message = “'Salam' de”
>>>k = 23
>>> pi = 3.14159
```

Bu misalda üç mənimsətmə əmri var. Birincisi **message** adlı yeni dəyişənə “‘Salam’ de” qiymətini mənimsədir. İkinci **k** dəyişəninə **23** qiymətini, üçüncü isə **pi** dəyişəninə sürüşkən vergüllü **3.14159** ədədini mənimsədir.

Mənimsətmə əməlini (=) bərabərlik ilə qarışdırmaq olmaz. Mənimsətmə əməliyyatı işarədən solda duran dəyişəni sağda duran qiymətlə əlaqələndirir. Adi riyaziyyatda **21=gun** qəbul ediləndirsə, Python dilində bu səhv verəcək. Çünki mənimsətmə işarəsindən solda mütləq dəyişən olmalıdır.

print əmri də dəyişənlərlə işləyir.

```
>>> print message            # və ya print (message)
'Salam' de
>>> print k                  # və ya print (k)
23
>>> print pi                 # və ya print (pi)
3.14159
```

Hər üç halda **print** əmrinin yerinə yetməsinin nəticəsi dəyişənin qiymətinin çapıdır. Dəyişənlərin də tipləri olur.

```
>>> type(message)
<type 'str'>
>>> type(k)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Dəyişənlərin adları və açar sözlər

Digər proqramlaşma dillərində olduğu kimi Python dilində də dəyişənlərə ad verərkən bəzi halları nəzərə almaq lazımdır. **Dəyişənin adı** ixtiyarı uzunluqda ola bilər. Adın tərkibində hərf və rəqəmlər ola bilər. Əsas şərt odur ki, ad hərflə başlamalıdır. Adlarda reqistr (böyük və kiçik hərflər) vacib rol oynayır. **Kitab** və **kitab** – bunlar tamamilə fərqli dəyişənlərdir. Çünki birinci hərflər müxtəlif reqistrlərdə verilib.

Adlarda aşağı xətdən (_) tez-tez istifadə edilir. Bir neçə adın birləşməsindaən əmələ gələn adlar buna misaldır. Məsələn, **My_name**.

Dəyişənə qadağan olunmuş ad versək sintaksis səhv olacaq.

```
>>> 55nomre = "böyük darvaza"
SyntaxError: invalid syntax
>>> nomre$ = 1955
SyntaxError: invalid syntax
>>> class = "Kompüter Elmi 103"
SyntaxError: invalid syntax
```

55nomre adı rəqəmlə başladığına görə səhvdir.

nomre$** adında icazə verilməyən **$ simvolu var.

Bəs **class** adı necə? Bu ad Python dilinin açar sözlərindən biridir. Açar sözlər Python dilinin elementidir və onlar adi adlarda istifadə edilə bilməz.

Aşağıdakı cədvəldə Python proqramlaşdırma dilindəki 31 açar sözlər verilib:

and	**as**	**assert**	**break**	**class**	**continue**
def	**del**	**elif**	**else**	**except**	**exec**
finally	**for**	**from**	**global**	**if**	**import**
in	**is**	**lambda**	**not**	**or**	**pass**
print	**raise**	**return**	**try**	**while**	**with**
yield					

Bu siyahını həmişə əl altında saxlayın. Pythonun interpretatoru dəyişənlərdən hansınınsa adının səhv olduğunu bildirsə, həmin adı bu siyahı ilə yoxlaya bilərsiniz.

Funksiyalar

Proqramlaşdırmada müəyyən bir fəaliyyəti yerinə yetirən adlanmış əmrlər (kodlar, skriptlər) ardıcıllığı **funksiya** adlanır. Funksiyalar adlanmış və anonim, **def** ifadəsi ilə yaradılan olur. Bundan başqa **return** və **lambda** funksiyaları da mövcuddur.

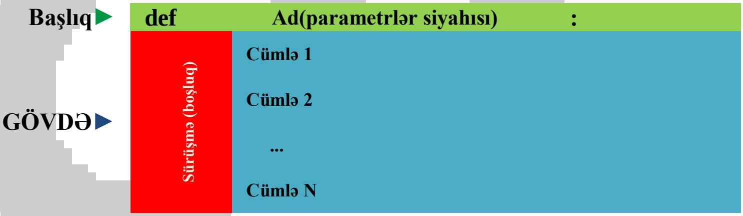
Python dilində funksiya deidkdə arqument qəbul edib qiymət qaytaran obyekt başa düşülür və adətən belə təyin olunur:

def ad (parametrlər siyahısı):
cümlələr

Yaradılan funksiyaya istənilən ad (rezervlənmiş açar sözlərdən savayı) vermək olar. Əgər funksiyanın işi üçün hansısa informasiyanın ötürülməsinə ehtiyac varsa, onda bu halda parametrlər siyahısı vasitəsilə istifadəçi bu informasiyanı funksiyaya ötürür.

Funksiya daxilində istənilən sayda əmr ola bilər. Burada əsas vacib şərt odur ki, onlar **def** sözünə nəzərən sağa sürüşməlidirlər. Çox vaxt bu sürüşmə adətən 4 **probel** olur. Funksiyanın təyin olunması cümləsi bu funksiyanı təşkil edən bizim gördüyümüz əmrlər içində birincidir. Onlar belə bir sxemlə qurulur:

1. **Başlıq** – **def** açar sözü ilə başlayır və ikinöqtə (**“:”**) ilə bitir.
2. **Gövdə**– Python dilində yazılmış və başlığa nəzərən eyni (adətən 4 probel) sürüşməsi olan azı bir və ya daha çox cümlədən ibarətdir.



Sxemdən göründüyü kimi başlıqda açar söz **def** istifadə edilir. Ondan sonra ad gəlir. Addan sonra mötərizə içində parametrlər siyahısı gəlir. Sonda iki nöqtə qoyulur. Parametrlər siyahısı boş, ya da istənilən sayda ola bilər. Boş və ya dolu olmağından asılı olmayaraq mötərizə olmalıdır.

Əvvəlcə parametri olmayan funsiyaya aid bir misala baxaq.

```
def Yeni_setir():
        print()
print ("Birinci sətir.")
print ("İkinci sətir.")
Yeni_setir()
print ("Üçüncü sətir.")
```

Birinci iki sətirdə funksiya verilmişdir. Funksiyanın adı **Yeni_setir** - dir. Parametri yoxdur. Gövdəsində boş sətir çap edən bircə ədəd **print()** əmri var.

Daha sonra əsas proqramın kodları gəlir. Əsas proqram 4 koddan ibarətdir. Birinci 2 kod 2 sətir mətn çap edir. Üçüncü kod boş sətir çap etmək üçün funksiyaya müraciət edir. Və nəhayət dördüncü kod yenə də mətn çap edir.

Proqramın işinin nəticəsi aşağıda verilib:

```
Birinci sətir.
İkinci sətir.

Üçüncü sətir.
```

İki sətir arasındakı boş sətir **Yeni_setir()** funksiyasının işinin nəticəsidir. Əgər boş sətirlərin sayını daha çox etmək istəyiriksə, onda əsas proqramda lazım olan qədər **Yeni_setir()** funksiyasını yazmaq lazımdır. Məsələn:

```
print ("Birinci sətir.")
print ("İkinci sətir.")
Yeni_setir()
Yeni_setir()
Yeni_setir()
Yeni_setir()
print ("Üçüncü sətir.")
```

Bu proqram fraqmentinin işinin nəticəsi belə olacaq:

```
Birinci sətir.
İkinci sətir.




Üçüncü sətir.
```

Əlbəttə ki, biz birdən artıq boş sətir çap edən funksiya yaradıb proqram kodunun həcmini azalda bilərik. Bu funksiyanın yaradılmasını Siz sərbəst yerinə yetirə bilərsiniz.

Başqa bir sadə funksiya təyin edək:

```
def add(x, y):
   return x + y
```

return əmri bildirir ki, qiymәti qaytarmaq lazımdır. Bizim misalda funksiya **x** və **y** arqumentlərinin cəmini qaytarır.

İnterpretator rejimində funksiya belə çağıralacaq:

```
>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'
```

Funksiya yaratmağın nə üstünlükləri var?

- Bir qrup proqram kodu olan əmrlər qrupuna bir ad vermək olar. Sonradan həmin adla həmin əmrlər qrupuna dəfələrlə müraciət etmək olar.

- Bu ümumi proqramın həcminin kifayət qədər kiçilməsinə gətirib çıxarır.

- Proqram daha gözə gəlimli və anlaşıqlı olur.

- Bir funksiya digər bir funksiyaya müraciət edə bilər.

Funksiyanın daxilindəki cümlə (lər) funksiya çağırılmayana qədər yerinə yetirilməyəcəklər.

Funksiya yerinə yetirilməyənə qədər o təyin olunmalıdır. Başqa sözlə, əsas proqramda funksiya birinci dəfə çağırılana qədər o təyin olunmalıdır. Növbəti sxem bunu əks etdirir:

Əsas proqramın mətni
Funksiya 1-in təyini
Funksiya 2-nin təyini
...
Funksiya N-in təyin
Əsas proqramın kodunun başlaşğıcı **Funksiya 1-ə müraciət** **...** **Funksiya 2-yə müraciət** **...** **Funksiya N-ə müraciət** **...** **Əsas proqramın kodunun sonu**

Öncədə dedik ki, funksiya yerinə yetirilənə qədər təyin edilməlidir. Bunun üçün proqram daxilindəki əmrlərin (kodların, skriptlərin) yerinə yetirilmə ardıcıllığını bilmək lazımdır. Bunu proqramın fəlsəfəsi kimi də başa düşmək olar. Proqramdakı əmrlərin yerinə yetirilmə ardıcıllığı **icra axını** adlanır.

Yerinə yetmə həmişə proqramın birinci sətrindən başlayır. Proqramın əmrləri hər yuxarıdan aşağı gedəndə bir dəfə yerinə yetirilir.

Funksiyanın təyini proqramın icra axınını dəyişmir. Amma funksiya daxili əmrləri funksiyaya müraciət olana qədər yerinə yetirilmir. Bəzən bir funksiya daxilində başqa bir funksiyanı təyin etmək olar (hərçənd ki, belə hallar çox nadir olur). Bu halda daxili funksiya xarici funksiya çağırılmayana qədər yerinə yetirilə bilməz.

Funksiyanın çağırılmasını icra axınından sapma (yayınma) kimi başa düşmək olar. Növbəti əmrə keçmək əvəzinə çağılıran funksiyanın gövdəsinin birinci əmrinə müraciət olunur və

funksiyanın gövdəsi yerinə yetirilir. Funksiyanın icrası bitdikdən sonra idarə funksiyaya müraciət olunan nöqtəyə qaytarılır və icranın davamı oradan davam etdirilir.

Göründüyü kimi prinsip çox sadədir. Bəs bir funksiya digər funksiyaya müraciət edən halda necə olur? Burada da Python dili cari funksiyaya girişin haradan olduğunu yadda saxlayır və hər dəfə funksiya işini bitirdikdən sonra proqram işini cari funksiya çağılmamışdan əvvəlki yerdən davam edir. Proqramın sonu çatdıqda o icrasını bitirir.

Bütün bu dediklərimizin fəlsəfəsi nədən ibarətdir? Proqramı oxuyarkən yuxarıdan aşağıya doğru oxumaq yanaşmasından qəti surətdə imtina edin. Yalnız və yalnız icra axını üzrə proqramın işini izləyin.

Yuxarıda nümunə üçün baxdığımız funksiyadan fərqli olaraq əksər funksiyalara emal etmək üçün arqumentlər lazım olur. Məsələn, hansısa ədədin mütləq qiymətini tapmaq üçün həmin ədədi göstərmək lazımdır. Python dilində mütləq qiyməti hesablamaq üçün daxili **abs** funksiyası mövcuddur.

```
>>> abs(23)
23
>>> abs(-23)
23
```

Bu misalda **abs** funksiyasının arqumentləri kimi **23** və **-23** ədədləri götürülmüşdür. Bəzi funksiyalar birdən çox arqumentlə işləyirlər. Məsələn, daxili **pow** funksiyası **2** arqumentlə işləyir, əsas və qüvvət. Funksiyaya ötürülən arqumentlər funksiya daxilində **parametr** adlanan dəyişənlərə mənimsədilir.

```
>>> pow(2, 5)
32
>>> pow(9, 4)
6561
```

Növbəti 2 funksiya ilə tanış olaq: **max** və **min**. Bu funksiyalar çox arqumentlidir. Mötərizə daxilində bir-biri ilə vergüllə ayrılmaqla çox sayda arqument vermək olar. **max** funksiyası bu arqumentlərin içindən maksimumu, **min** funksiyası isə minimumu seçib nəticə kimi qaytarır. Arqumentdə həm sadə qiymətlər, həm də ifadələr ola bilər.

```
>>> max(11, 23, 7, 11)
23
>>> min(11, 23, 7, 11)
7
>>> max(2 * 15, 4**3, 1024 - 100, 512**0)
924
```

İndi isə yeni bir parametrli funksiya yaradaq.

```
def print_melumat(par_sahe):
    print par_sahe, par_sahe
```

Bu funksiya yeganə arqumenti qəbul edir və onun qiymətini **par_sahe** adlı parametrə mənimsədir. Funksiya daxilində əvvəlcə parametrin qiyməti iki dəfə çap edilir və sonda sətrin sonu çap edilir.

```
>>> def print_melumat(par_sahe):
    print (par_sahe, par_sahe)
>>> print_melumat('Yoxlama')
Yoxlama Yoxlama
>>> print_melumat('5')
5 5
>>> print_melumat('3.14159')
3.14159 3.14159
```

Burada **par_sahe** parametrə bizim verdiyimiz addır. Bu ad ixtiyarı (açar sözlərdən savayı) ola bilər. Adı elə seçmək lazımdır ki, o təyinatı haqqında müəyyən informasiya daşısın.

Python redaktorunda bu proqram parçasını belə yazmaq olar:

```
def print_melumat(par_sahe):
    print (par_sahe, par_sahe)
print_melumat('Yoxlama')
print_melumat('5')
print_melumat('3.14159')
```

Proqramın işinin nəticəsi isə interpretatorda belə alınacaq:

```
Yoxlama Yoxlama
5 5
3.14159 3.14159
```

Biz bu misalda proqramın iki rejimdə (həm interpretator (interaktiv), həm də redaktor) yerinə yetirilməsinə baxdıq. İnteraktiv rejimdə müəyyən proqram parçalarını, funksiyaları test etmək rahatdır.

Importəmrindən istifadə etməklə skriptdə təyin edilmiş funksiyanı interpretatora yükləyib istifadə etmək olar. Tutaq ki, **print_melumat** funksiyası Pythonun kitabxanasında **cap.py** adı ilə yadda saxlanılıb. Bu zaman onu import edib istifadə etmək olar:

```
>>> from cap import *
>>> print_melumat('Yoxlama')
Yoxlama Yoxlama
>>> print_melumat('5')
5 5
>>> print_melumat('3.14159')
3.14159 3.14159
```

Müxtəlif tipli qiymətlər çap etmək üçün onu **print_melumat** funksiyasına ötürmək olar. Yuxarıdakı misalda birinci sətir, ikinci tam ədəd və sonda sürüşkən vergüllü ədəd çap edilib.

Daxili funksiyalarda olduğu kimi yaradılan funksiyalarda da ifadələrdən istifadə etmək olar:

```
>>> from cap import *
>>> print_melumat('Yoxlama'*4)
YoxlamaYoxlamaYoxlamaYoxlama
YoxlamaYoxlamaYoxlama
```

Bu misalda öncə **'Yoxlama'*4** ifadəsi hesablanır (**'YoxlamaYoxlamaYoxlamaYoxlama'**), sonra isə alınan qiymət çap üçün arqument kimi**print_melumat** funksiyasına ötürülür.

Riyaziyyatda olduğu kimi Python dilinin funksiyalarını kombinə etmək olar, yəni bir funksiyanın nəticəsindən digər funksiyaya müraciət edəndə arqument kimi istifadə etmək olar.

```
>>> from cap import *
>>> print_melumat(max(7, 5, abs(-8), 6))
8 8
```

Arqument kimi dəyişəndən istifadə etmək olar:

```
>>> deyishen="Azərbaycan! "
>>> print_melumat(deyishen)
Azərbaycan!  Azərbaycan!
```

Funksiyalardan danışarkən lokal və qlobal dəyişənləri qeyd etmək lazımdır. Bildiyimiz kimi dəyişənlər təsir dairəsinə görə 2 yerə bölünürlər: lokal və qlobal dəyişənlər. Əvvəlcə gəlin görünmə sahəsi anlayışı ilə tanış olaq.

Görünmə sahəsi (ing:**scope**) – dedikdə proqramlaşdırmada elə sahə nəzərdə tutulur ki, onun dairəsində cari anda dəyişənlərin və funksiyaların adlarına müraciət etmək olsun. Başqa sözlə görünmə sahəsi dəyişənlərin təyin olunduğu və onların axtarıldığı yerdir.

Proqramda hər hansı bir dəyişəndən istifadə ediləndə interpretator həmişə bu adı adlar saxlanan sahədə-adlar fəzasında yaradır, dəyişir və ya axtarır.Biz adın proqram koduna nəzərən axtarılmasından danışanda **görünmə sahəsi** dedikdə adlar fəzası: proqram kodunda ada qiymət mənimsədilən yer başa düşülür. Bu yer konkret adın proqram kodu üçün görünmə sahəsini təyin edir.

def açar sözü daxilində təyin edilən adlar yalnız onun təyin olunduğu funksiya daxilində görünür. Bu funksiyadan kənarda bu ada müraciət etmək olmur. **def** daxilində təyin edilən adlar həmin funksiyadan kənarda təyin edilən adlarla (eyni adlı olsa da) konfliktdə olmur. Çünki onlar adları eyni olsa da müxtəlif dəyişənlərdir.

Hansısa bir funksiya daxilində dəyişənə mənimsətmə olmuşsa, onda həmin dəyişən təyin olunduğu funksiya daxilində **lokal dəyişən** olacaq.

Əgər dəyişənə mənimsətmə bütün **def** təlimatlarından kənarda təyin edilmişsə, onda o bütün fayl üçün **qlobal dəyişən** olacaq.

Dediyimiz kimi, əgər Siz funksiya daxilində **lokal dəyişən** yaradırsınızsa, o yalnız funksiya daxilində təsir dairəsinə malikdir. Yəni həmin adlı dəyişənə funksiyadan kənarda müraciət etmək olmaz. Məsələn:

```
def sahe_dubl(dey1, dey2):
cem = dey1 + dey2
   print_melumat(cem)
```

Bu funksiya 2 arqument qəbul edir. Daha sonra onları konkatenasiya edilir və **cem** dəyişəninə mənimsədilir. Sonda **cem** dəyişəninin qiyməti çap edilir. Gəlin funksiyanın işini proqramda izləyək.

```
>>> def sahe_dubl(dey1, dey2):
   cem = dey1 + dey2
   print_melumat(cem)

>>> dey1 = "İşləməyən, "
>>> dey2 = "Dişləməz."
>>> sahe_dubl(dey1, dey2)
İşləməyən, Dişləməz. İşləməyən, Dişləməz.
```

sahe_dubl funksiyası işini bitirdikdən sonra **cem** dəyişəni ləğv edilir. Onu təkrar ayrıca çap etmək istəsək səhv verəcək.

```
>>> print cem
SyntaxError: Missing parentheses in call to 'print'
```

Dediklərimizi yekunlaşdıraq. Hansısa proqram kodu yazılmış fayl daxilində hansısa ada qiymət mənimsədilmişsə və bundan sonra yeni funksiya təyin edilib və funksiya daxilində həmin ada yeni qiymət mənimsədilmişsə, bu halda funksiyadan çıxan kimi ikinci mənimsətmə öz təsirini itirir və birinci mənimsətmə yenə qüvvəyə minir:

```
>>> a = 23
>>> def func():
     a = 31
>>> print (a)
23
```

Gördüyünüz misalda funksiyadan əvvəl **a** dəyişəninə **23** ədədi mənimsədilmişdir. Daha sonra yeni funksiya təyin edilib və onun daxilində həmin dəyişənə **31** qiymətimənimsədilib. Funksiyadan sonra yazılmış **print (a)** əmri yerinə yetiriləndə isə çapda **23** qiyməti alınır. Bu halda **a** dəyişəni bu fayl üçün qlobaldır. Qlobal görünüş sahəsi bütün faylı əhatə edir, lokal isə yalnız cari funksiyanı.

Bundan başqa qlobal dəyişəni müəyyən etmək üçün **global** əmrindən istifadə edilir. Daha yüksək səviyyəli faylda təyin edilmiş dəyişəndən aşağı səviyyədə onu təyin etmədən də istifadə etmək olar.

Daxili funksiyalar

Python dili bir sıra daxili funksiya və tiplərə malikdir. Aşağıdakı cədvəldə onlar əlifba sırası ilə verilib.

Daxili funksiyalar (Built-in Functions)				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	_
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Gördünüz kimi Python dilində daxili funksiyalar çoxdur. Rahatlıq üçün onları kateqoriyalara bölürlər:

1. Tipləri və sinifləri çevirən funksiyalar: **all, any, ascii, bin,str, repr, int, list, tuple, float, complex, dict, super, bool, object**

Bu kateqoriyadan olan funksiya və siniflər verilənləri çevirmək üçün nəzərdə tutulub. Python dilinin köhnə versiyalarında veriləni uyğun tipə çevirmək eyniadlı funksiyadan istifadə edilirdi. Pythonun yeni versiyalarında belə funksiyaların rolunu daxili siniflərin adları oynayır.

2. Ədədlər və sətirlərlə işləyən funksiyalar: **abs, divmod, ord, pow, len, chr, hex, oct, round**

Bu kateqoriyadan olan funksiyalar ədədi və sətir arqumentləri ilə işləyir.

3. Verilənləri emal edən funksiyalar:**sorted, filter, zip, range, map, max, min, iter, enumerate, sum**

Bu kateqoriyadan olan funksiyalar funksional proqramlaşdırmada istifadə edilir.

4. Xassələri müəyyənləşdirən funksiyalar: **hash, id, callable, issubclass, isinstance, type**

Bu kateqoriyadan olan funksiyalar obyektlərin daxili atributlarına və digər xassələrinə çıxışı təmin edir.

5. Daxili strukturlara müraciət edən funksiyalar: **locals, globals, vars, dir**

Python dilinin müasir realizasiyasında qlobal və lokal dəyişənlər globals() və locals() funksiyaları sayəsində lüğət formasında əlçatandır. Burada bir şeyi demək lazımdır ki, bu lüğətlərə nəsə yazmaq məsləhət deyil.

6. Kompilyasiya və icra funksiyaları: **eval, exec,__import__, compile**

Bu kateqoriyadan olan funksiyalar onlara ötürülən ifadələri hesablayırlar.

7. Daxiletmə-çıxış funksiyaları:**input, open**

8. Atributlarla işləyən funksiyalar:**getattr, setattr, delattr, hasattr**

Python dilində obyektlərin atributları ola bilər.

9. Siniflərin metodlarını "bəzəyən" funksiyalar:**staticmethod, classmethod, property**

Bu kateqoriyadan olan funksiyalar obyekt-yönümlü proqramlaşdırmada istifadə edilir.

10. Digər funksiyalar: **buffer, slice**

MƏSLƏHƏT:
Funksiyanin təyinatını müəyyən etmək üçün Python dilinin interpretator rejimində aşağıdakı **help**və ya **print** funksiyalarından istifadə etmək kifayətdir:

>>>**help**(funksiyanın adı) məsələn, **help(len)**

və ya

>>>**print**(funksiyanın adı**.__doc__**) məsələn, **print (len.__doc__)**

Sonrakı bölmələrdə bu daxili funksiyaların əksəriyyəti haqqında daha geniş məlumat veriləcək. Əgər bu kitabda hansısa daxili funksiya haqqında məlumat tapmasanız, bu halda **help()** funksiyasının köməyindən bəhrələnə bilərsiniz.

Standart kitabxananın icmalı

Digər proqramlaşdırma dilləri kimi Python dilinin də standart kitabxanaları mövcuddur. Python dilində standart kitabxananın modullarını şərti olaraq aşağıdakı mövzular üzrə qruplara bölmək olar.

1. **Yerinə yetirilmə dövrünün servisləri.**Modullar: sys, atexit, copy, traceback, math, cmath, random, time, calendar, datetime, sets, array, struct, itertools, locale, gettext.

2. **Layihələşdirmə dövrünün dəstəklənməsi.**Modullar: pdb, hotshot, profile, unittest, pydoc. docutils, distutils paketləri.

3. **ƏS ilə qarşılıqlı əlaqə (fayllar, proseslər).**Modullar: os, os.path, getopt, glob, popen2, shutil, select, signal, stat, tempfile.

4. **Mətnlərin emalı**. Modullar: string, re, StringIO, codecs, difflib, mmap, sgmllib, htmllib, htmlentitydefs. xml paketi.

5. **Çoxaxınlı hesablamalar**. Modullar: threading, thread, Queue.

6. **Verilənlərin yadda saxlanılması. Arxivləşmə**. Modullar: pickle, shelve, anydbm, gdbm, gzip, zlib, zipfile, bz2, csv, tarfile.

7. **Platforma-asılı modullar**. **UNIX üçün**: commands, pwd, grp, fcntl, resource, termios, readline, rlcompleter. **Windows üçün**: msvcrt, _winreg, winsound.

8. **Şəbəkənin dəstəklənməsi. İnternet protokolları.**Modullar: cgi, Cookie, urllib, urlparse, httplib, smtplib, poplib, telnetlib, socket, asyncore. Serverlərin nümunələri: SocketServer, BaseHTTPServer, xmlrpclib, asynchat.

9. **İnternetin dəstəklənməsi. Verilənlərin formatı**.Modullar: quopri, uu, base64, binhex, binascii, rfc822, mimetools, MimeWriter, multifile, mailbox. email paketi.

10. **Python özü haqqında**.Modullar: parser, symbol, token, keyword, inspect, tokenize, pyclbr, py_compile, compileall, dis, compiler.

11. **Qrafik interfeys**. Tkinter modulu.

Modullar. Onların yaradılması. import və from əmrləri ilə qoşulma

Funksiyaları öyrənərkən aydın oldu ki, proqram kodunu təkrar istifadə etmək üçün onu funksiyaya necə yerləşdirmək olar. Bəs müxtəlif funksiyaları digər proqramlarda istifadə etmək üçün nə etmək lazımdır? Əlbəttə ki, moduldan istifadə etməklə bu mümkündür.

Əgər Siz interpretatordan çıxıb yenidən ona daxil olsanız, onda Sizin təyin etdiyiniz bütün adlar (funksiya və dəyişənlər) itəcək. Buna görə də əgər Siz daha iri həcmli proqram yazmaq istəyirsinizsə, bu halda Python redaktorundan istifadə edə bilərsiniz. Yazılmış proqramı fayldan daxiletmə rejimində interpretatorda icra edin. (Yazılmış proqram mətni bəzən **ssenari** və ya **skript** adlanır.) Əgər Sizin proqramınızın mətninin həcmi çox böyükdürsə, onda istismarın rahatlığı üçün onu bir neçə fayla ayırmaq olar. Bundan başqa Siz hansısa bir faydalı proqram kodundan və ya qlobal dəyişəndən digər proqramlarda istifadə etmək istəyirsinizsə, bu halda həmin təlimatı ayrıca fayl şəklində yadda saxlaya bilərsiniz.

Python dilində əmrlər yazılmış istənilən fayl ***modul (module)*** adlanır. Modullardan Python dilinin digər proqramlarında istifadə etmək olar. Modulda təyin olunan kod, qlobal dəyişənlər, sinif, funksiya və obyektlər digər modullarda, ya da baş modulda import oluna bilər.

Modul yaratmaq üçün müxtəif üsullar var. Amma bunlardan ən sadəsi funksiya və dəyişənləri özündə saxlayan və tipi **.py**olan fayl yaratmaqdır. Modul daxilində modulun adına (sətir kimi) **_name_** adlı qlobal dəyişən kimi müraciət etmək olar.

Paket (package) – adi modullar toplusudur, daha doğrusu modulların iearxiyalı ağacabənzər strukturda yığmaq üçün istifadə edilən modul növüdür. Faktiki olaraq paketlər özündə altqovluqları saxlayan qovluqdur. Bu qovluq və altqovluqların daxilində modullar saxlanılır. Modullara müraciət nöqtələrdən istifadə etməklə həyata keçirilir.

Tutaq ki, **package** paketi aşağıdakı struktura malikdir:

```
        package/
  __init__.py
  module1.py
  module2.py
  module3.py
 package1/
    __init__.py
    module11.py
    module12.py
    module13.py
 package2/
    __init__.py
    module21.py
    module22.py
    module23.py
 package3/
    __init__.py
    module31.py
    module32.py
    module33.py
```

Əgər **module33.py** faylında yerləşən **myfunc()** funksiyasına müraciət etmək istəyirsinizsə, onda onu import etmək üçün aşağıdakı əmrdən istifadə etmək olar:

importpackage.package3.module33.myfunc()

Paket olan hər bir qovluq **__init__.py** faylına malik olmalıdır. Bu faylın kodu ilk dəfə import olunanda avtomatik icra edilir.

import a.b.c

yazılışında ardıcıl olaraq aşağıdakılar yerinə yetəcək:

import a

import a.b

import a.b.c

Bunu yadda saxlamaq lazımdır.

Python dilindəki proqramlarda istifadə edilən modullar öz mənşəyinə görə adi (Python dilində yaradılanlar) və genişlənmiş, yəni başqa proqramlaşma dilində yazılanlardır (adətən C və C^{++} dilində). İstifadəçi nöqteyi-nəzərindən onlar sürətinə görə fərqlənə bilərlər. Bəzən elə olur ki, standart kitabxanada modulun iki variantı olur: Python və C dillərində.

Məsələn, pickle и cPickle modulları. Adətən Python dilində yazılmış modul genişlənmiş modula nəzərən daha çevik olur.

“Python kitabxanası” bir neçə növ komponentdən ibarətdir.

Birinci qrup komponentlər dilin “nüvəsi” hesab edilirlər. Həmin qrupa ədədlər və ya siyahılar daxildir. Kitabxanaya həmçinin daxili funksiyalar da daxildir.

Amma kitabxananın əsas hissəsi digər modullardan ibarətdir. Bu modullardan bəziləri **C** dilində yazılıb və Python dilinin interpretatoruna quraşdırılıb. Digərləri Python dilində yazılıb və ilkin proqram kodlarına import olunublar. Bəzi modullar Python dilinin yüksək səviyyəli imkanları üçün interfeys rolu oynayır. Digər qrup internetlə iş üçün nəzərdə tutulub.

Əlbəttə ki, komponentlər haqqında çox danışmaq olar. Texniki nöqteyi-nəzərdən izah etsək görərik ki, modul ayrıca fayldır. İnterpretator modulun faylında mənimsədilmə aparılan bütün adları özündə saxlayan modulun obyektini yaradır. Başqa sözlə, modullar sadəcə olaraq adlar fəzası (adlar yaradılan yerlər) və modulda olan və onun atributları adlanan adlardır.

Modulun standart kitabxanadan importu

Modulu **import** əmri vasitəsilə import etmək olar. Məsələn, **os** modulunu import etməklə cari qobluq haqqında məlumat ala bilərik:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
```

Import açar sözündən modulun adı yazılır. Bir əmrlə bir neçə modulu qoşmaq olar (Hərçənd ki, bunu etmək məsləhət görülmür. Çünki belə yazılış proqram mətnin oxunaqlığını aşağı salır). **time** və **random**modullarını import edək:

```
>>> import time, random
>>> time.time()
1444758496.707366
>>> random.random()
0.6532282574830747
```

Modulu import etdikdən sonra onun adı dəyişən olur. Bu ad vasitəsilə modulun atributlarına müraciət etmək olar. Məsələn: **math** modulunun **e** sabitinə müraciət edə bilərik:

```
>>> import math
>>> math.e
2.718281828459045
```

Əgər modulun verilmiş atributu tapılmasa, **AttributeError** istisnası yaranacaq. Əgər import üçün modul tapılmasa, onda **ImportError** istisnasıyaranacaq.

Qoşma addan (alias) istifadə

Əgər modulun adı çox uzundursa, və ya ad Sizin xoşunuza gəlmirsə, onda Siz **as** açar sözü vasitəsilə qoşma ad (ləqəb) yarada bilərsiniz.

```
>>> import math as m
>>> m.e
2.718281828459045
```

Siz artıq modulun bütün atributlarına **m** dəyişəni vasitəsilə müraciət edə bilərsiniz. Təkrar **import math** əmri verilənə qədər bu proqramda **math** modulunun bütün atributlarına **m** dəyişəni vasitəsilə müraciət ediləcək.

from əmri

Modulun müəyyən atributlarını **from** əmri vasitəsilə qoşmaq olar. Onun bir neçə formatı var:

```
from <Modulun adı> import <Atribut 1> [ as <Qoşma ad 1> ], [<Atribut 2> [ as
<Qoşma ad 2> ] ...]
from <Modulun adı> import *
```

Birinci format modulun yalnız Sizin verdiyiniz atributlarını qoşmağa imkan verir. Uzun adlar üçün **as** açar sözündən sonra qoşma vermək olar.

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(5.7)
6
```

Oxunaqlığı yaxşılaşdırmaq üçün import olunan atributları bir neçə sətirdə yazmaq olar:

```
>>>>>> from math import (sin, cos,
                         tan, atan)
```

Əmrin ikinci formatı imkan verir ki, modulun bütün (demək olar ki, bütün) dəyişənləri qoşulsun. Misal üçün **sys** modulunun bütün atributlarını import edirik:

```
>>> from sys import *
>>> version
'3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)]'
>>> version_info
sys.version_info(major=3, minor=4, micro=3, releaselevel='final', serial=0)
```

Bu variantda bütün atributlar import olunur.

Python dilində öz modulunu yaratmaq

İndi bir modul yaradaq. Daxilində müəyyən funksiyalar təyin olunan **mymodule.py** adlı fayl yaradaq:

```
def salam():
    print('Salam, dostum!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

İndi həmin qovluqda **main.py** adlı daha bir fayl yaradaq:

```
import mymodule

mymodule.salam()
print(mymodule.fib(10))
```

Ekrana aşağıdakı məlumat çıxacaq:

```
Salam, dostum!
55
```

Sizi yenə təbrik etmək olar! Artıq Siz öz **modulunuzu** da yaratdınız!

Modulu necə adlandırmaq?

Bir şeyi nəzərə almaq lazımdır ki, Sizin yaratdığınız moduldan ölkə daxilində və xaricində digərləri də dəyişən kimi istifadə edəcək. Ona görə də modula ad verəndə aşağıdakıları nəzərə almaq lazımdır:

- açar sözlərdən istifadə etmək olmaz;
- ad rəqəmlə başlaya bilməz;
- yaratdığınız modula mövcud funksiyaların adını qoymaq olmaz Bu gələcəkdə problemlər yarada bilər.

Modulu harada yadda saxlamalı?

Yaratdığınız modulu elə yerdə saxlamalısınız ki, gələcəkdə onu istifadəçi tapa bilsin. Modulların axtarılması yolu **sys.path** dəyişənində yerləşir. Onun məzmununa baxmaq üçün aşağıdakı əmrləri vermək lazımdır:

```
>>> import sys
>>> sys.path
['', 'C:\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\SYSTEM32\\python34.zip',
'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-
packages']
```

Gördüyünüz kimi, **sys.path** dəyişəninin məzmununa cari qovluq daxildir (yəni modulu əsas proqramla eyni qovluqda saxlamaq olar). Bundan başqa ora **python** dilinin quraşdırıldığı qovluqlar da daxildir. **sys.path** dəyişəninin əllə də dəyişmək olar. Bu halda biz modulu istidiyimz yerdə yerləşdirə bilərik.

Modulu sərbəst proqram kimi istifadə etmək olarmı?

Bu mümkündür. Ancaq bir şeyi yadda saxlamaq lazımdır ki, modulu import edəndə onun kodu bütövlükdə yerinə yetirilir. Bu o deməkdir ki, əgər proqram nəyisə çap edirsə, o import edilərkən də eyni şey çap ediləcək. Bu problemi aradan qaldırmaq üçün yoxlamaq lazımdır ki, modul adi proqram kimi yerinə yetirilir, yoxsa import olunub. Bunu **__name__** dəyişəni vasitəsilə etmək olar. Bu dəyişən bütün proqramlarda təyin edilib. Əgər skript baş proqram kimi yerinə yetirilirsə, o **"__main__"** qiyməti alır, import olunubsa, qiyməti ad olur. Məsələn, **mymodule.py** belə ola bilər:

```
def hello():
    print('Salam, dost!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b

if __name__ == "__main__":
    hello()
    for i in range(10):
        print(fib(i))
```

sys modulu

sys modulu əməliyyat sistemi ilə yüksək səviyyəli əlaqədə olur. Bu modula Python dilinin interpretatoru ilə əlaqədə olan funksiya və sabitlər daxildir. **sys** moduluna həmçinin aşağıdakı dəyişənlər daxildir:

argv — Proqrama (skriptə) ötürülən arqumentlərin siyahısı. Birinci arqument skriptin faylının ünvanını göstərən tam marşrutdur.

byteorder — 'little' və ya 'big' platformasının bayt ardıcıllığı.

copyright — Python interpretatorunun istehsalı (**copyright**) haqqında məlumat olan sətir.

dllhandle — Python interpretatorunun DLL deskriptoru haqqında məlumat.

exc_info() — Sonuncu baş verən istisna haqqında məlumatı qaytarır. Əgər istisna yoxdursa, boş sətir qaytarır, əks halda aşağıdakı sahələrə malik sabitlər siyahısı qaytarır:

- **type** – istisnanın tipi (onun sinfinin adı);
- **value** – istisna obyekti;
- **traceback** – stek, skriptin vəziyyəti haqqında informasiya.

executable — İnterpretatorun yolu.

exit([arg]) — Sistemə **arg** çıxış kodunu ötürməklə çıxış.

flags — Obyekt atributlar şəklində interpretatora bayraqlar haqqında informasiya ötürür. Məsələn, **sys.flags.debug** sazlanma rejimi haqqında məlumat verir.

getdefaultencoding() — Susma halında Unicodun kodlaşdırılması.

maxunicode — Unicodda maksimal simvol. Bu simvol sistemdə quraşdırılmış Unicodun versiyasından asılıdır.

modules — Yüklənmiş modulların siyahısı.

platform — Platformanın identifikatoru, məsələn, **'win32'.**

path — Əməliyyat sistemini axtarılması üçün marşrutlar siyahısı.

stdin, stdout, stderr — Girişin, çıxışın və səhvlərin standart axını.

version — Versiyanı göstərən sətir.

version_info — İnterpretatorun versiyasını major, minor, micro, releaselevel, və serial formatında əks etdirən sabitlər siyahısı. Məsələn, Python 3.4.3 versiyası üçün sys.version_info(major=3, minor=4, micro=3, releaselevel='final', serial=0).

os modulu

os modulu əməliyyat sistemləri ilə işləmək üçün çoxlu sayda funksiyalar təklif edir. Bu funksiyalar əsasən ƏS-dən asılı deyillər, deməli *daşınandırlar*.

os.name – Əməliyyat sisteminin adı. Məsələn: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

os.environ – Dəyişənlər ətraf mühitinin lüğəti. Dəyişəndir (ətraf mühitinin dəyişənlərini əlavə etmək və silmək olar).

os.getpid() – Prosesin cari **id**-i.

os.uname()–ƏS haqqında informasiya. Növbəti atributlu obyekt qaytarır: sysname - əməliyyat sisteminin adı, nodename –şəbəkədəki maşının adı, release - reliz, version - versiya, machine –maşının identifikatoru.

os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)–cari istifadəçidə obyektə çıxışın yoxlanılması. Bayraqlar: os.F_OK –obyekt mövcuddur,os.R_OK –oxunmağa açıqdır, os.W_OK –yazılmağa açıqdır, os.X_OK –icraya açıqdır.

os.chdir(path) – cari direktoriyanın (qovluğun) dəyişməsi.

os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True) – obyektə çıxış hüququnun dəyişməsi (mode –səkkizlik ədəd).

os.getcwd() – Direktoriyanın cari işi.

os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)–Sərt müraciət yaradır.

os.listdir(path=".") – Qovluqdakı fayl və direktoriyaların siyahısı.

os.mkdir(path, mode=0o777, *, dir_fd=None) – Direktoriya yaradır. Əgər direktoriya mövcuddursa, OSError.

os.makedirs(path, mode=0o777, exist_ok=False) – Aralıq direktoriya yaratmaqla direktoriya yaradır.

os.remove(path, *, dir_fd=None) – Fayla olan yolu silir.

os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None) – Fayl və ya direktoriyanın adını **src** –dən **dst**-yə dəyişir.

os.renames(old, new) – Aralıq direktoriya yaratmaqla **old**adını**new** adına dəyişir.

os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None) – **src** adından **dst** adına məcburi dəyişir.

os.rmdir(path, *, dir_fd=None) – Boş direktoriyanı silir.

os.removedirs(path) – Direktoriyanı silir, daha sonra üst direktoriyanı silir və bunu onlar boşalana qədər rekursiv edir.

os.symlink(source, link_name, target_is_directory=False, *, dir_fd=None) – Obyektə simvolik müraciət yaradır.

os.truncate(path, length) – Faylı **length** uzunluğuna qədər kəsir.

os.utime(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True) – Fayla sonuncu müraciət və onun dəyişilmə vaxtının modofikasiyası. Ya **times** – korteji (saniyələrlə vaxt), ya da**ns** - kortej (nanosaniyələrlə vaxt).

os.walk(top, topdown=True, onerror=None, followlinks=False) – direktoriyalar (qovluqlar) ağacında yuxarıdan aşağı (əgər topdown=True), və aşağıdan yuxarı (topdown=False) adların generasiyası. Hər bir qovluq üçün **walk** funksiyası kortej qaytarır.

os.system(command) – Sistem əmrini yerinə yetirir.

os.urandom(n) – **n** təsadüfi bayt. Bu funksiyadan kriptoqrafik məqsədlər üçün istifadə etmək olar.

os.path – Marşrutla bağlı bəzi faydalı funksiyaları realizə edir.

time modulu

Pythondilində **time** modulu vaxtla işləmək üçün nəzərdə tutulub.

time.altzone –Saat qurşağının sıfırıncı meridiana nəzərən qərbə dəyişdirilməsi. Saat qurşağı şərqdədirsə, sürüşmə mənfi olacaq.

time.asctime([t])–Korteji və ya **struct_time**-ni "Thu Sep 27 16:42:37 2012" görünüşlü sətrə çevirir. Əgər arqument yoxdursa, cari vaxt götürülür.

time.clock()–Unix-də cari vaxtı qaytarır. Windows əməliyyat sistemində bu funksiyanın ilk dəfə çağırıldığı andan ötən vaxtı qaytarır.

time.ctime([san])–Əsrin əvvəlindən saniyələrlə verilən vaxtı "Thu Sep 27 16:42:37 2012" görünüşlü sətrə çevirir.

time.daylight - Yay və ya qış vaxtı (DST) təyin edilibsə, 0 (sıfır) deyil.

time.gmtime([san]) - Əsrin əvvəlindən saniyələrlə verilən vaxtı **struct_time**-ya çevirir. Bu halda DST bayraq həmişə sıfıra bərabər olur.

time.localtime([san])–**gmtime** kimi, amma DST bayraqlı.

time.mktime(t) - Korteji və ya **struct_time**-ni əsrin əvvəlindən olan saniyələrə çevirir. **time.localtime** funksiyasına əks.

time.sleep(san)–Proqramın icrasını verilən miqdarda saniyə qədər saxlayır.

time.strftime(format, [t]) - Korteji və ya **struct_time**-ni aşağıdakı formata görə sətrə çevirir:

Format	Təyinatı
%a	Həftənin gününün qısa adı
%A	Həftənin gününün tam adı
%b	Ayın qısa adı
%B	Ayın tam adı
%c	Tarix və vaxt
%d	Ayın günü [01,31]
%H	Saat (24-saatlıq format) [00,23]
%I	Saat (12-saatlıq format) [01,12]
%j	İlin günü [001,366]
%m	Ayın nömrəsi [01,12]

Format	Təyinatı
%M	Dəqiqələrin ədədi [00,59]
%p	Günortaya qədər və ya sonra (12-saatlıq formatda)
%S	Saniyələrin ədədi [00,61] (2)
%U	İldə həftənin nömrəsi (sıfırıncı həftə bazar günündən başlayır) [00,53]
%w	Günün həftədəki nömrəsi [0(Sunday),6]
%W	Həftənin ildəki nömrəsi (sıfırıncı həftə bazar günündən başlayır) [00,53]
%x	Tarix
%X	Vaxt
%y	Əsrsiz il [00,99]
%Y	Əsrin əvvəlindən il
%Z	Vaxt (saat) zonası
%%	'%' işarəsi

time.strptime(sətir [, format])–Formata uyğun vaxtı əks etdirən sətrin araşdırılması. **struct_time** qiymətini qaytarır. Susma halında format belə olacaq: **"%a %b %d %H:%M:%S %Y".**

time.struct_time sinfi –Vaxtın qiymətinin ardıcıllığınıntipi. Kortej interfeysinə malikdir. İndeksə və ya ada görə müraciət etmək olar.

tm_year

tm_mon

tm_mday

tm_hour

tm_min

tm_sec

tm_wday

tm_yday

tm_isdst

time.time()–Əsrin əvvəlindən saniyələrlə ifadə edilən vaxt.

time.timezone –Yerli saat qurşağının sıfırıncı meridiandan qərbə saniyələrlə sürüşdürülməsi. Əgər saat qurğagı şərqdədirsə, sürüşmə mənfi olur.

time.tzname –İki sətirdən ibarət kortej: birinci–saat qurğaşının DST adı, ikinci –yerli saat qurşağının adı.

random modulu

random modulu təsadüfi ədədlərin, hərflərin generasiya edilməsi, ardıcıllığın elementlərinin təsadüfi seçilməsi funksiyaları təqdim edir.

random.seed([X], version=2)–Təsadüfi ədədlərin generatorunun inisializasiyası. Əgər **X**verilməyibsə, sistem vaxtından istifadə edilir.

random.getstate()–Generatorun daxili vəziyyəti.

random.setstate(state)–Generatorun daxili vəziyyətini bərpa edir. **state** parametri **getstate()** funksiyası vasitəsilə alınır.

random.getrandbits(N)–**N** təsadüfi bit qaytarır.

random.randrange(start, stop, step)–Ardıcıllıqdan təsadüfi seçilən ədədi qaytarır.

random.randint(A, B)–Təsadüfi tam **N** ədədi, **A ≤ N ≤ B**.

random.choice(sequence)–Boş olmayan ardıcıllığın təsadüfi ədədi.

random.shuffle(sequence, [rand])–Ardıcıllığı qarışdırır (ardıcıllıq özü dəyişir). Buna görə funksiya dəyişilməyən obyektlər üçün işləmir.

random.sample(population, k) - **population** ardıcıllığından **k** uzunluqlu siyahı.

random.random()–**0** və **1** arasında təsadüfi ədəd.

random.uniform(A, B) - Sürüşkən vergüllü təsadüfi **N** ədədi. **A ≤ N ≤ B** (və ya **B ≤ N ≤ A**).

random.triangular(low, high, mode) - Sürüşkən vergüllü təsadüfi **N** ədədi.**low ≤ N ≤ high**. **mode** - paylama.

random.betavariate(alpha, beta)–Beta-paylama. **alpha>0, beta>0**. **0** və **1** arasında arasında təsadüfi ədəd qaytarır.

array modulu. Python dilində massiv

Pytho dilində dəyişən kimi ayrıca massiv təyin edilməmişdir. Bunun üçün **array** modulu massiv yaratmaq üçün nəzərdə tutulub. Massivlər siyahılara çox oxşayır. Amma siyahıdan verilənlərin tipi və hər bir elementin ölçüsünə məhdudiyyəti ilə fərqlənir.

Massivin ölçüsü və tipi o yaradılarkən təyin edilir və aşağıdakı qiymətləri alır:

Tipin kodu	C-də tip	Python-da tip	Baytlarla minimal ölçü
'b'	signed char	int	1
'B'	unsigned char	int	1
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2

'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Massivlər dəyişilə biləndir. Massivlər bütün siyahı metodları (indekslənmə, kəsik, hasil, iterasiya) və digər metodlarla işləyir.

array.typecode - TypeCode simvolundan massiv yaradılarkən istifadə edilir.

array.itemsize –Massivdə bir elementin baytlarla ölçüsü.

array.append(x)–Massivin sonuna elementin əlavə edilməsi.

array.buffer_info() - Kortej (yaddaşın oyuğu, uzunluq). Aşağı səviyyəli əməliyyatlar üçün faydalıdır.

array.byteswap()–Massivin hər bir elementində baytların gəlmə ardıcıllığını dəyişir. Kompüterdə başqa ardıcıllıqla yazılmış verilənlərin oxunması zamanı faydalıdır.

array.count(x)–**x**-ın massivdə neçə dəfə yerləşməsini qaytarır.

array.extend(iter)–Elementlərin obyektdən massivə əlavə edilməsi.

array.frombytes(b)–Baytlar massivindən array massivi düzəldir. Baytların sayı massivdəki bir elementin ölçüsünə bölünən olmalıdır.

array.fromfile(F, N)–Fayldan **N** element oxuyur və massivin sonuna yazır. Fayl binar (ikilik) oxunuş üçün açılmalıdır. Əgər **N**–dən az element varsa, **EOFError** istisnası generasiya olunur. Amma daxil edilən elementlər massivə əlavə edilir.

array.fromlist(siyahı)–Elementlərin siyahıdan daxil edilməsi.

array.index(x)–x-ın massivə birinci dəfə daxil edilməsi.

array.insert(n, x)–Massivdə **n** nömrəsindən əvvəl **x** qiymətli yeni bənd daxil edir. Mənfi qiymət massivin sonuna nəzərən qəbul edilir.

array.pop(i)–Massivdən **i**-ci elementi silir və ya qaytarır. Susma halında sonuncu element silinir.

array.remove(x)–Massivdən olan **x**-ın birinci daxil olmasını silir.

array.reverse()–Massivdəki elementlərin tərs düzülüşü.

array.tobytes()–Baytlara çevirmə.

array.tofile(f)–Massivi açıq fayla yazır.

array.tolist()–Massivi siyahıya çevirir.

turtle modulu

İcraçı bağa rəsm sahəsində durduğu cari nöqtəyə nəzərən nisbi əmrləri (irəli, geri və s.) idarə edir. İcraçı özündən sonra iz buraxa bilən qələmdir. Qələmi qaldırmaq (bu halda hərəkət zamanı iz qalmayacaq) və endirmək olar. Bundan başqa qələmin qalınlığını və rəngini müəyyən etmək olar. İcraçının bütün bu funksiyaları **turtle** modulu vasitəsilə təmin edilir.

Əmr	Şərh	Nümunə
up()	Qələmin qaldırılması.	turtle .up()
down()	Qələmin endirilməsi	turtle .down()
goto(x,y)	Qələmin rəsm çəkmə pəncərəsində **(x, y)** koordinatına yerini dəyişməli (getməli).	turtle .goto(50,20)
color ('rəng)	Qələmin rənginin qiymətini rəng sətrində müəyyən edilimiş kimi təyin edir.	turtle.color('blue ') turtle.color ('#0000ff')
width(n)	Qələmin qalınlığını n-ə uyğun ekranın nöqtələri qalınlığında etməli.	turtle .width(3)
forward(n)	n nöqtə irəli	turtle . forward(100)
backward(n)	n nöqtə geri	turtle .backward(100)
right (k)	k vahid sağa dönməli (saat əqrəbi istiqamətində)	turtle . right (75)
left (k)	k vahid sola dönməli (saat əqrəbinin əksi istiqamətində)	turtle . left (45)
radians ()	Bucağın ölçü vadini radianla götürməli	turtle . radians ()
degrees ()	Bucağın ölçü vadini dərəcə ilə götürməli(susma halında)	turtle .degrees ()
circle (r)	Qələmin cari nöqtəsindən başlayaraq |r| radiuslu çevrə çəkməli. Əgər r müsbətdirsə, saat əqrəbi istiqamətində, mənfidirsə, saat əqrəbinin əksi istiqamətində halda çəkməli.	turtle . circle (40) turtle . circle (−50)
circle (r ,k)	|r| radiuslu və k vahid bucaqlı qövs çəkməli. circle () əmrinin variant.	turtle . circle (40,45) turtle . circle (−50,275)
fill (flag)	flag parametrinin qiymətindən asılı olaraq oblastı rəngləmə (flag=1) və rəngləməmə (flag=0) rejimi qoşulur. Susmahalındaqoşulmama rejimi işləyir.	Kpyr: turtle . fill (1) turtle . circle (−50) turtle . fill (0)
write ('sətir')	Mətni qələmin cari mövqeyində yazmalı	turtle . write ('Koordinatın başlanğıcı!')
tracer (flag)	flag parametrinin qiymətindən asılı olaraq qələmin işarəsi görünür (flag=1) və ya görünmür (flag=0). Susma halında görünmə rejimi işləyir.	turtle . tracer (0)
clear ()	Rəsm çəkmə sahəsinin təmizlənməsi	turtle . clear (0)

Anonim funksiyalar, lambda ifadəsi

Anonim funksiyalar yalnız bir ifadədən ibarət olur, amma onlar daha sürətli işləyirlər. Anonim funksiyalar **lambda** ifadəsi vasitəsilə yaradılırlar. Bundan başqa def func() ifadəsində olduğu kimi onları dəyişənə mənimsətmək vacib deyil:

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

Lambdafunksiyası adi funksiyalardan fərqli olaraq **return** ifadəsini istəmir. Yerdə qalan hər şey oxşardır.

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Metodlar

Metod obyektə (məsələn, sətirə) tətbiq edilən funksiyadır. Metod aşağıdakı formada çağırılır:

Obyektin_adı.Metodun_adı(parametrlər)

Məsələn: **Setir.find("a")**. Bu metod **Setir adlı** sətirə bir parametrli (**"a"**) **find** metodunun tətbiqidir.

Metodlar haqqında sonrakı bölmələrdə bir daha danışacağıq.

Python dilinin sintaksisi

Python dilinin sintaksisi dilin özü kimi sadədir.

Sintaksis

- Sətrin sonu əmrin sonu kimi başa düşülür (nöqtə-vergül qoymağa ehtiyac yoxdur).
- Bir-birinə tabe olan əmrlər girintilərin (sağa sürüşmələrin) ölçüsünə görə blokda birləşirlər. Girinti ixtiyari ola bilər, təki tabe olan bir blokun daxilində girinti eyni olsun. Oxunuşu da nəzərə almaq lazımdır. 1 probel qoymaq məsləhət deyil. Yaxşısı budur ki, bir neçə (məsələn: 4) probeldən (ya da tabulyasiyadan) istifadə edin.
- Python dilində tabe olan əmrlər eyni şablona uyğun yazılır. Əsas əmr iki nöqtə ilə bitirsə və ondan sonra tabe olan əmr bloku gəlirsə, onda o əsas əmri olan sətrin altından girintili vəziyyətdə (əvvəlində 4-5 probel olmaq) yerləşir.

```
Əsas əmr:
        Daxil olan əmrlər bloku
```

Xüsusi hallar

- Bəzən bir neçə əmri bir sətirdə yazmaq olar. Bu zaman onlar nöqtə-vergüllə ayrılırlar:

```
a = 1; b = 2; print(a, b)
```

Ancaq buna çox da aludə olmayın! Proqram kodunun oxunuşu rahat olmalıdır.

- Bir əmri bir neçə sətirdə yazmaq olar. Bunun üçün əmri ümumi ixtiyari mötərizəyə almaq lazımdır:

```
a = 1; b = 2; c = 3; d = 4; print(a, b, c, d)
if (a == 1 and b == 2 and
c == 3 and d == 4):  # İki nöqtə yaddan çıxarılmamalıdır
        print('spam' * 3)
```

- Əgər tabe əmrin özünün tabe olan əmrləri yoxdursa, onda tabe əmrin gövdəsi əsas əmrin olduğu sətirdə yazıla bilər. Məsələn:

if x > y: print(x)

Dinamik tipləşmə

Python dilində dinamik tipləşdirmədən istifadə edilir. Bu o deməkdir ki, dəyişənləri proqram mətnində öncədən elan etmək lazım deyil. Dəyişənə qiymət mənimsədiləndə dəyişənin tipi avtomatik müəyyənləşdirilir. Eyni bir dəyişən proqramın müxtəlf hissələrində tam ədəd, sonra həqiqi ədəd, daha sonra isə simvol sətri, siyahı, kortej və ya lüğət ola bilər: Məsələn:

```
A = 123 # tam ədəd
A = 1.7 # həqiqi ədədi
A = "Salam dostum!" # sətir
A = [1, 2, 3, 4, 5] # siyahı
A = (1, "Aysu", 3) # kortej
A = {"Aysu": 8, "Ömər": 2, "Qalib": 42} # lüğət
```

Bu xüsusiyyət bir tərəfdən proqramçını "qandaldan azad edir", digər tərəfdən onun üzərinə məsuliyyət qoyur. Bu mövzuya sonralar yenə qayıdacağıq. İndi isə Python dilində dəyişənlərin tipləri ilə tanış olaq.

Python dilində tiplər

Python dilində dəyişənlərin çox tipi var. Daha çox vacib tiplər (növlər) aşağıdakılardır:

- məntiqi – iki qiymətdən (True-Həqiqi və ya False-Yalan) birini alır,
- ədədlər – tam (1 və 2), sürüşkən vergüllü (2.4 və 3.6), kəsr (1/3 və 2/5) və kompleks ədədlər,
- sətirlər – UNICOD simvolları ardıcıllığı, məsələn, HTML-sənəd,
- baytlar və baytlar massivi – məsələn, JPEG formatlı fayl,
- siyahılar – nizamlanmış qiymətlər ardıcıllığı,
- kortejlər – nizamlanmış dəyişilməyən (sabit) qiymətlər ardıcıllığı,
- çoxluqlar (set) – nizamlanmamış qiymətlər toplusu,
- lüğətlər – açar-qiymət növlü nizamlanmamış çütlüklər toplusu.

Məntiqi dəyişənlər və onlar üzərində əməliyyatlar

Python proqramlaşdırma dilində məntiqi qiymətlər iki məntiqi sabitlə, True (Həqiqi) və False (Yalan) ilə təsvir olunur. Məntiqi qiymət məntiqi əməliyyatlar və məntiqi ifadələrin hesablanması nəticəsində alınır.

Əsas məntiqi əməliyyatlar və ifadələr

Əməliyyat və ifadələr	Təsviri
>	"Böyükdür" şərti (məsələn, a>b)
<	"Kiçikdir" şərti (məsələn, a<b)
==	Bərabərlik şərti (a bərabərdir b şərtini yoxlayırıq)
!=	Bərabərsizlik şərti (a bərabər deyil b şərtini yoxlayırıq)
not x	İnkar (x şərti yerinə yetirilmir)
x and y	Məntiqi "HƏ" (vurma). x and y şərti ödənmək üçün eyni vaxtda x və y şərti ödənməlidir.

Əməliyyat və ifadələr	Təsviri
x or y	Məntiqi "YOX" (toplama). x or y şərti ödənmək üçün x və y şərtlərindən biri ödənməlidir.
x in A	x elementinin A çoxluğuna (strukturuna) aid olması yoxlanılır.
a < x < b	(x>a) and (x<b) şərtinə ekvivalentdir.

Məntiqi əməliyyatın hesablanması nəticəsində məntiqi qiymət alınır. Məsələn, **if** operatoru gözləyir ki, ifadəni hesablama nəticəsində alınan nəticə məntiqi qiymət olacaq. Belə hissələr məntiqi kontekst adlanır. Məntiqi kontekstdə istənilən ifadəni istifadə etmək olar. Python istənilən halda onun həqiqiliyini yoxlayacaq.

Şərti proqramdan bir kodlaşma parçasına baxaq:

```
if size < 0:
   print('ədəd mənfi ola bilməz')
```

Burada tipi tam ədəd olan **size** və **0** verilmişdir. Onlar arasında < işarəsi ədədi operatordur. **size** < **0** ifadəsinin hesablanması isə həmişə məntiqi qiymət olacaq. True

Python dilində məntiqi qiymət ədəd kimi götürülə bilər. True = 1 və False = 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> False/True
0.0
>>> True / False
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Burada sıfıra bölmə halı olduğu uçun xəbərdarlıq edilir.

Yaxşısı budur belə hallara yol verməyin. Yoxsa özünüzü çətinliyə salarsınız.

Python dilində həqiqiliyin yoxlanması

- 0-a bərabər olmayan istənilən ədəd və ya boş olmayan obyekt - **həqiqi**dir.
- 0-a bərabər ədədlər, boş obyektlər və None qiyməti - **yalan**dır.
- Müqayisə əməliyyatları verilənlər strukturuna rekursiv tətbiq edilir.

- Müqayisə əməliyyatları **True** və ya **False** qaytarır.
- **and** və **or** məntiqi operatorları həqiqi və ya yalan obyekt-operand qaytarır.

Məntiqi operatorların qiymətləri:

X	Y	not X	not Y	X or Y	X and Y
0	0	1	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	1	1

Ədədlər: tam, həqiqi, kompleks

Ədədlər çox əsrətamis bir şeydir. Onlar o qədər çoxdurlar ki, həmişə yeni nəsə seçmək olar. Yuxarıda deyildiyi kimi, Python dilində dinamik tipləşdirmədən istifadə edilir. Yəni dəyişənin (o cümlədən ədədlərin) tipinin təyin edilməsinə ehtiyac yoxdur. Bu işi Python ədəddə nöqtənin olub-olmamasından asılı olaraq avtomatik edir. Python dilində ədədlərin aşağıdakı tipləri var:

- adi tam ədədlər (int tipi),
- uzun tam ədədlər (long tipi),
- həqiqi ədədlər (float tipi),
- kompleks ədədlər.

```
>>> type(1)
        <class 'int'>
>>> 1 + 1
        2
>>> 1 + 1.0
        2.0
>>> type(2.0)
        <class 'float'>
```

Python dilində ədədlər adi ədədlərdən fərqlənmirlər. Python dilində ədədlər üzərində adi riyazi əməliyyatlar aparmaq olur:

Ədədlər üzərində əməllər

Əməliyyat və ifadələr	Təsviri
x+y	Toplama
x-y	Çıxma
x*y	Vurma
x/y	Bölmə. Diqqət! Əgər x və y tam ədədlərdirsə, onda nəticə tam ədəd olacaq! Nəticənin həqiqi ədəd olması üçün ədədlərdən biri həqiqi ədəd olmalıdır. Məsələn: 100/8 →12, amma 100/8.0 →12.5 olacaq.
x//y	Tam bölmə (nəticə tam ədəddir). Əgər hər iki ədəd həqiqidirsə, onda nəticədə kəsr hissəsi sıfıra bərabər olan həqiqi ədəd alınır. Məsələn: 100//8 → 12 101.8//12.5 → 8.0 (müqayisə üçün 101.8/12.5 → 8.1440000000000001)
x%y	x və y tam ədədlərinin bölünməsindən alınan qalıq. Məsələn: 10%4 → 2
xy**	Qüvvətə yüksəltmək (x üstü y). Həqiqi ədədlər üçün də işləyir. Məsələn: 2**3 → 8 2.3**(−3.5) → 0.05419417057580235
-x	Ədədin işarəsini dəyişir.
abs(x)	Ədədin modulu.
divmod(x, y)	(x // y, x % y) cütü. Bölmə nəticəsində alınan tam və qalıq hissəni verir. divmod(23,4)→(5, 3)
pow(x, y[, z])	x^y tapılır. Əgər z yoxdursa, alınan nəticə qəbul edilir. Z olan zaman alınan nəticənin z-ə bölünməsindən alınan qalıq götürülür. Bir növ **pow(x, y)%z** əmri işləyir.

Növbəti cədvəldə dəyişənlər üzərində aparılan əməliyyatın nəticəsinin tipinin operantların (əməliyyatda iştirak edən elementlər) tipindən necə asılı olması göstərilib:

I operant	II operant	Nəticə	Nümunə
TAM	**TAM**	**TAM**	100/8 →12
TAM	**HƏQİQİ**	**HƏQİQİ**	100/8.0 →12.5
HƏQİQİ	**TAM**	**HƏQİQİ**	100.0/8 →12.5
HƏQİQİ	**HƏQİQİ**	**HƏQİQİ**	100.0/8.0 →12.5

Tam ədədlər (int)

Python dilində tam ədədlərin 2 tipi var:

- **adi tam ədədlər (int tipi),**
- **uzun tam ədədlər (long tipi),**

Sadəcə olaraq bir şeyi yadda saxlamaq lazımdır ki, uzun tam ədədlərlə işləyəndə daha böyük yaddaş sərf edilir.

Python dilinin interpretatorunda bir neçə misala baxaq:

```
>>>255+44
299
>>>5*3
15
>>>23/7
3.2857142857142856
>>>23//7
3
>>>23%5
3
>>>3**4
81
>>> divmod(23,4)
(5, 3)
>>>pow(3,4)
81
>>>pow(3,4,23)
12
>>>3**160
21847450052839212624230656502990235142567050104912751880812823948662932355201
```

Bit əmrləri

Tam ədədlər üzərində aşağıdakı bit əmrlərini yerinə yetirmək olar. Əməliyyat ədədin yaddaşdakı ikilik yazılışı üzərində aparılır.

x | y	Bitlərlə *OR (və ya)*
x ^ y	Bitlərlə *müstəsna OR (və ya)*
x & y	Bitlərlə *AND (və)*
x << n	Bitlərlə sola sürüşmə
x >> y	Bitlərlə sağa sürüşmə
~x	Bitlərin inversiyası (əksi, neqativi). 0-lar 1-lərlə, 1-lər 0-larla əvəz edilir

Məsələn: ikilik say sistemində verilmiş ədədi bir mərtəbə vahidi qədər artırmaq üçün onu bir bit sola sürüşdürmək lazımdır.

Həqiqi ədədlər (float)

Həqiqi ədədlərlə də tam ədədlərlə apardığımız əməliyyatları aparmaq olar. Ancaq həqiqi ədədlərin kompüterdə yaddaşda yerləşməsi formasına görə (o dəqiq olmur) nəticədə səhv ola bilər. Məsələn:

```
>>> 1+1+1+1+1+1+1+1+1+1
10
>>>0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
0.9999999999999999
```

Misaldan göründüyü kimi ikinci əmrdə **0.1** ədədinin tipi həqiqidir (float). Ona görə də cəmdə gözlədiyimiz **1.0** nəticəsi əvəzinə **0.9999999999999999** nəticəsi alınır. Daha dəqiq nəticə almaq üçün digər obyektlərdən(məsələn, decimal və fraction) istifadə etmək məsləhətdir.

Bundan başqa həqiqi ədədlər uzun ədədlərlə işləmir:

```
>>>a=3**1000
>>>a+0.1
Traceback (most recent call last):
File "", line 1, in
OverflowError: int too large to convert to float
```

Sadə misallara baxaq:

```
>>>c=250
>>>d=14.3
>>>c+d
264.3
>>>p=abs(d-c)# Ədədin modulu
```

```
>>>print(p)
235.7
>>>round(p)# Yuvarlaqlaşdırma
236
```

Kompleks ədədlər (complex)

Python dilində kompleks ədədlərlə də iş nəzərdə tutulub:

```
>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> y = complex(3, 4)
>>> print(y)
(3+4j)
>>> z = x + y
>>> print(z)
(4+6j)
>>> z = x * y
>>> print(z)
(-5+10j)
>>> z = x / y
>>> print(z)
(0.44+0.08j)
>>> print(x.imag)  # Xəyali hissə
2.0
>>> print(x.real)  # Həqiqi hissə
1.0
```

İş prosesində bəzən ədədləri bir tipdən digərinə çevirmək lazım gəlir. Python dilində həqiqi ədədləri tam və əksinə çevirən funksiyalar təyin olunub. Bunlar **int()** və **float()**. Məsələn,

```
>>> int (12.6)
     12
>>> int (-12.6)
     -12
>>> float (12)
    12.0
```

Misallardan göründüyü kimi, **int()** funksiyası ədədin kəsr hissəsini atır və əgər müsbətdirsə, tam hissəni yuvarlaqlaşdırmır, yox əgər ədəd mənfidirsə, tam hissəni artım istiqamətində yuvarlaqlaşdırır. **float ()** əmri isə tam ədədin sonuna qiyməti sıfır olan kəsr hissə əlavə edir.

Say sistemləri

Hələ məktəb İnformatika kursundan bilirik ki, kompüter ikilik say sistemi ilə işləyir. Bundan başqa proqramlaşdırmada səkkizlik və onaltılıq say sistemlərindən də istifadə edilir. Aşağıda bir nümunə verilib:

Onluq say sistemi	İkilik say sistemi	Səkkizlik say sistemi	Onaltılıq say sistemi
23	**10111**	**27**	**17**

Bəzən verilmiş ədədi bir say sistemindən digərinə çevirməyə ehtiyac olur. Python dilində bu məqsədlə uyğun funksiyalar mövcuddur:

- **int**([object], [say sisteminin əsası]) –əsası 2 ilə 36 arasında olan say sistemində dəyişən verilmiş ədədi onluq say sistemində tam ədədə çevirir. Susma halında say sistemi onluq götürülür.
- **bin**(x) –tam ədədi ikilik say sistemindəki ədədə çevirir.
- **hex**(x) - tam ədədi onaltılıq say sistemindəki ədədə çevirir.
- **oct**(x) - tam ədədi səkkizlik say sistemindəki ədədə çevirir.

Nümunə:

```
>>> a = int('23') # Sətri ədədə çevirir
>>> b = int('23.5')  # Sətir tam ədəd deyil
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module>
b = int('23.5')  # Sətir tam ədəd deyil
ValueError: invalid literal for int() with base 10: '23.5'
>>>c=int(23.5)# Sürüşkən vergüllü ədədin kəsr hissəsini atır
>>>print(a,c)
2323
>>> bin(23)
'0b10111'
>>> oct (23)
'0o27'
>>> hex(23)
'0x17'
>>>0b10111# ədədi sabitləri belə yazmaq olar
23
>>>int('10111',2)
23
>>>int('0b10111',2)
23
```

Python dilində sətirlərlə iş. Literallar

Python dilində sətir verilənlərin çox çətin tipidir. Sətir deyəndə mətn formasında informasiyanın yadda saxlanılması və təqdim edilməsi üçün istifadə edilən nizamlanmış, UNICOD-da kodlaşmış ixtiyari simvollar ardıcıllığıdır. Buna görə də sətirlərlə mətn formasında verilən istənilən hər şeylə işləmək olar. Sətirlərapastrof (tək dırnaq) və ya dırnaq (cüt dırnaq) arasında yazılır. Burada iki variantın olmasının əsas səbəbi odur ki, mətnə apastrof və dırnaq simvollarını salmaq mümkün olsun.

```
S = 'sınaq"dır'
S = "sınaq'dır"
```

Sətrin apastrof və ya dırnaq arasında yazılmasının fərqi yoxdur, ikisi də eynidir. Aşağıda verilmiş iki yazılış eynidir (ekvivalentdir).

“Bu bir sətirdir”* ≡ *‘Bu bir sətirdir’

Simvollar sətrinə adətən sətrin literalları deyilir.

Python dilində sətirlərlə işləmək çox rahatdır. Bir neçə sətirlər literalı mövcuddur.

Python dilində ekranlaşmış ardıcıllıqdan (xidməti simvollardan) istifadə edilir.

UNICODda verilmiş bütün simvolları klaviaturdan daxil etmək olmur. Ekranlaşmış ardıcıllıq klaviaturdan daxil edilə bilməyən simvolları sətrə əlavə etməyə imkan verir.

Ekranlaşmış ardıcıllıq	Təyinatı
\n	Növbəti sətrə keçmək
\a	Zəng
\b	Kəsmək (bölmək)
\f	Növbəti səhifəyə keçid
\r	Sətir başlarının (karetkanın) qaytarılması
\t	Üfqi tabulyasiya
\v	Şaquli tabulyasiya
\N{id}	UNİCOD verilənlər bazasının ID identifikatoru
\uhhhh	16-bitlik UNICOD simvolu 16-lıqda təsviri
\Uhhhh…	32-bitlik UNICOD simvolun 32-likdə təsviri
\xhh	Simvolun 16-lıqda qiyməti
\ooo	Simvolun 8-likdə qiyməti
\0	Null simvolu (sətrin sonu əlaməti deyil)

Məsələn:

```
>>> A = ["alma", "armud","heyva","nar"]
>>> print (A)
['alma', 'armud', 'heyva', 'nar']
>>> print (A[0], '\n', A[1], '\t', A[2], '\b', A[3], '\a')
alma
 armud           heyva  nar
```

Əgər açılan dırnaqdan əvvəl "**r**" simvolu (hansı reqistrdə olmasının fərqi yoxdur) durursa, ekranlaşma mexanizmi ləğv edilir. Belə sətrə "**çiy sətir**" və ya **"formatlanmamış sətir"**deyilir.

Məsələn (öndə təyin etdiimiz siyahı üçün):

```
>>> print (A[0], r'\n', A[1], '\t', A[2], r'\b', A[3], '\a')
alma \n armud                    heyva \b nar
```

"Çiy" sətir tərsinə çəpinə xətlə bitə bilməz. Belə halın həlli yolu belədir:

```
S = r'\n\n\\'[:-1]
S = r'\n\n' + '\\'
S = '\\n\\n'
```

Əgər böyük mətni bir neçə sətirdə yazmaq istəyiriksə, bu zaman **üçdırnaqdan (üç apastrofdan)** istifadə etmək olar. Belə sətrin daxilində dırnaq və apastrof ola bilər. Əsas odur ki, iki üçdırnaq ardıcıl gəlməsin.

```
>>> c = '''bu çox böyük
... sətirdir, çox sətirli
... mətn bloku'''
>>> c
'bu çox böyük\n... sətirdir, çox sətirli\n... mətn bloku'
>>> print(c)
bu çox böyük
... sətirdir, çox sətirli
... mətn bloku
```

Sətirlər. Sətirlər üçün funksiya və metodlar (üsullar)

Python dilində sətrin daxilində olan simvolların (siyahının və kortejlərin elementlərinin) sıra nömrəsi aşağıdakı sxemə əsasən müəyyən edilir. "**Qarabağ**" sətri üçün nömrələmə belə olacaq:

Q	a	r	a	b	a	ğ
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

Baza əməliyyatları

Əməliyyat	Yazılış	Nəticə	Şərh
Konkatenasiya (toplama)	>>> S1="Qara" >>> S2="bağ" >>> print (S1+S2)	Qarabağ	
Sətrin təkrarlanması (dubl edilməsi)	>>> print("bara" * 3)	barabarabara	
Sətrin uzunluğu (len funksiyası)	>>>len("Qarabağ")	7	
İndeksə görə keçid (müraciət)	>>>m="Qarabağ" >>>m[0] >>> m[6] >>> m[-1] >>> m[-7] >>> m[4] >>> m[-3] >>> m[-0] >>> m[7] Traceback (most recent call last): File "<pyshell#7>", line 1, in <module> m[7] IndexError: string index out of range	 'Q' 'ğ' 'ğ' 'Q' 'b' 'b' 'Q' Səhvdir	Python dilində mənfi indeksə görə də keçid etmək olar. Bu zaman saymaq sətrin sonundan başlayacaq. İndeksə görə keçiddən siyahılarda və kortejlərdə də istifadə etmək olar.
Kəsiyin alınması [X:Y]. X – kəsiyin başlanğıcı, Y – kəsiyin sonu. Y saylı simvol kəsiyə aid olmur. Susma halında birinci indeks **0**-a, ikinci isə sətrin uzunluğuna bərabər olur.	>>> m="Qarabağ" >>> m[1:6] >>> m[2:-2] >>> m[:4] >>>m[1:] >>>m[:] >>> m[::-1] >>> m[3:5:-1] >>> m[2::2] >>> m[3:5:1] >>> m[3:5:3]	 'araba' 'rab' 'Qara' 'arabağ' 'Qarabağ' 'ğabaraQ' '' 'rbğ' 'ab' 'a'	Üçüncü ədədi addım kimi vermək olar. Kəsiklərdən siyahılarda və kortejlərdə də istifadə etmək olar.

Üsullardan istifadə edərkən bir şeyi nəzərə almaq lazımdır ki, Python dilində sətirlər dəyişilməyən ardıcıllıqdırlar. Yəni bütün sətrin elementini ardıcıllıq kimi (indeksə görə) dəyişmək olmaz. Yalnız yeni sətir kimi yenidən yaratmaq olar. Belə bir misala baxaq:

```
>>> s = 'qaymaq'
>>> s
'qaymaq'
>>> s[2] = 'r'
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    s[2] = 'r'
TypeError: 'str' object does not support item assignment
>>> s = s[0:2] + 'r' + s[3:]
>>> s
'qarmaq'
```

Misaldan göründüyü kimi s sətrinə **“qaymaq”** mətni mənimsədilir. Sonra ardıcıllıq kimi 2-ci hərfi dəyişmək istəsək səhv kimi qəbul edilir. İkinci hərfi dəyişmək üçün həmin yerə qədər olan kəsiyi, dəyişmək istədiyimiz qiyməti və həmin yerdən sonra gələn kəsiyi konkatenasiya etmək yolu ilə məqsədimizə çata bilərik. Misal dediyimiz üsulla **s** sətrindəki **“qaymaq”** sözü bir hərf dəyişilməklə **“qarmaq”** sözünə çevrilib.

Beləliklə, aydın olur ki, sətirlərlə işləyən bütün üsullar (metodlar) yeni sətir yaradırlar. Ona görə də sonda onu dəyişənə mənimsətmək lazımdır.

İndi isə sətirlər üçün digər funksiya və üsullarla (metodlarla) tanış olaq.

Sətirlərin funksiyaları və metodları (üsulları)

Funksiya və metodlar	Təyinatı
Setir = b"byte"	Bayt sətri
Setir.find(str,[start],[end])	**str** altsətrini **Setir** sətrinin daxilində axtarır. Birinci daxilolmanın nömrəsini və ya **-1** verir. >>> Setir ="sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit" >>> Setir 'sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit' >>> Setir.find("aaaa",0,55) 5 >>> Setir.find("bbbb",0,55) -1
Setir.rfind(str,start],[end])	**str** altsətrini **Setir** sətrinin daxilində axtarır. Sonuncu daxilolmanın nömrəsini və ya **-1** verir. >>> Setir ="sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit" >>> Setir 'sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit' >>> Setir.rfind("aaaa",0,55) 43 >>> Setir.rfind("bbbb",0,55) -1
Setir.index(str,[start], [end])	**str** altsətrini **Setir** sətrinin daxilində axtarır. Birinci daxilolmanın nömrəsini və ya ValueError səhvini verir. >>> Setir ="sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit" >>> Setir 'sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit'

Funksiya və metodlar	Təyinatı
	>>> Setir.index("aaaa",0,55) 5 >>> Setir.index("bbbb",0,55) Traceback (most recent call last): File "<pyshell#27>", line 1, in <module> Setir.index("bbbb",0,55) ValueError: substring not found
Setir.rindex(str,[start], [end])	**str** altsətrini **Setir** sətrinin daxilində axtarır. Sonuncu daxilolmanın nömrəsini və ya ValueError səhvini verir. >>> Setir ="sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit" >>> Setir 'sdasdaaaajkkjhkjkj hjjjhjhj jhhjhjhjhj jhjhaaaatuiit' >>> Setir.rindex("aaaa",0,55) 43 >>> Setir.rindex("bbbb",0,55) Traceback (most recent call last): File "<pyshell#29>", line 1, in <module> Setir.rindex("bbbb",0,55) ValueError: substring not found
Setir.replace(əvəz edilən şablon, əvəz edən şablon)	Şablonun əvəz edilməsi. >>> Setir="bazarbazarbazar" >>> Setir.replace("z", "h") 'baharbaharbahar'
Setir.split(simvol)	Sətri ayırıcıya görə bölür. >>> Setir="ora-bura" >>> Setir.split("-") ['ora', 'bura']
Setir.isdigit()	Sətir yalnız rəqəmlərdən ibarətdirsə, həqiqi, əks halda yalan qaytarır. >>> Setir="1234567890" >>> Setir.isdigit() True >>> Setir="abcdefgh" >>> Setir.isdigit() False >>> Setir="a1b2c3d4" >>> Setir.isdigit() False
Setir.isalpha()	Sətir yalnız hərflərdən ibarətdirsə, həqiqi, əks halda yalan qaytarır.

Funksiya və metodlar	Təyinatı
	>>> Setir="abcdefgh" >>> Setir.isalpha() True >>> Setir="1234567890" >>> Setir.isalpha() False >>> Setir="a1b2c3d4" >>> Setir.isalpha() False
Setir.isalnum()	Sətir yalnız rəqəmlərdən və ya hərflərdən ibarətdirsə, həqiqi, əks halda yalan qaytarır. >>> Setir="a1b2c3d4" >>> Setir.isalnum() True >>> Setir="1234567890" >>> Setir.isalnum() True >>> Setir="abcdefgh" >>> Setir.isalnum() True >>> Setir="a1b2=+-c3d4" >>> Setir.isalnum() False
Setir.islower()	Sətir yalnız alt reqistrdə olan simvollardan ibarətdirsə, həqiqi, əks halda yalan qaytarır. >>> Setir="abcdefgh" >>> Setir.islower() True >>> Setir="ABCDEFGH" >>> Setir.islower() False >>> Setir="AaBbCcDdEeFfGgHh" >>> Setir.islower() False
Setir.isupper()	Sətir yalnız üst reqistrdə olan simvollardan ibarətdirsə, həqiqi, əks halda yalan qaytarır. >>> Setir="ABCDEFGH" >>> Setir.isupper() True >>> Setir="abcdefgh" >>> Setir.isupper() False >>> Setir="AaBbCcDdEeFfGgHh" >>> Setir.isupper() False

Funksiya və metodlar	Təyinatı
Setir.isspace()	Sətir görünməyən simvollardan (probel, '\f' – növbəti səhifəyə keçid, '\n' – növbəti sətrə keçid, '\r' – karetkanın çevrilməsi, '\t' – üfqi tabulyasiya, '\v' – şaquli tabulyasiya) ibarətdirsə, həqiqi, əks halda yalan qaytarır. >>> Setir="\f\n\r\t\v" >>> Setir.isspace() True >>> Setir="Sətir görünməyən simvollardan \f\n\r\t\v ibarətdir" >>> Setir.isspace() False
Setir.istitle()	Sətirdəki sözlər böyük hərflərlə başlayırsa, həqiqi, əks halda yalan qaytarır. >>> Setir="Bu Bir Yoxlama Mətnidir." >>> Setir.istitle() True >>> Setir="Bu Bir Yoxlama mətnidir." >>> Setir.istitle() False
Setir.upper()	Sətri üst reqistrə çevirir. >>> Setir="abcdefgh" >>> Setir.upper() 'ABCDEFGH'
Setir.lower()	Sətri alt reqistrə çevirir. >>> Setir="ABCDEFGH" >>> Setir.lower() 'abcdefgh'
Setir.startswith(str)	**Setir** sətri **str**şablonu ilə başlayırsa, həqiqi, əks halda yalan qaytarır. >>> Setir="Azərbaycan" >>> Setir.startswith("Az") True >>> Setir.startswith("az") False
Setir.endswith(str)	**Setir** sətri **str** şablonu ilə qurtarırsa, həqiqi, əks halda yalan qaytarır. >>> Setir="Azərbaycan" >>> Setir.endswith("can") True >>> Setir.endswith("Can")

Funksiya və metodlar	Təyinatı
	False
Setir.join(siyahı)	Verilmiş siyahıdan ayırıcı **Setir** olmaqla yeni sətir yığmaq. >>> Setir="=+=" >>> Setir.join(["alma","armud","heyva","gilas"]) 'alma=+=armud=+=heyva=+=gilas'
ord(simvol)	Simvoldan ASCII kod almaq. >>> ord("a") 97
chr(ədəd)	ASCII koddan simvol almaq. >>> chr(97) 'a'
Setir.capitalize()	Sətrin birinci simvolunu üst reqistrə, digərlərini isə alt reqistrə çevirir. >>> Setir="bu Bir YOXLAMA mƏtnidir" >>> Setir.capitalize() 'Bu bir yoxlama mətnidir'
Setir.center(width, [fill])	**Setir** sətrini mərkəzə yerləşdirir və kənarlara **fill** simvolunu yazır. Əgər **fill** verilməmişsə, kənarlarda probel (boşluq) yazır. >>> Setir="Salam dostum!" >>> Setir.center(30, "=") '========Salam dostum!=========' >>> Setir.center(30,) ' Salam dostum! ' >>> Setir.center(30) ' Salam dostum! '
Setir.count(str, [start],[end])	**str** altsətrinin (start, end) diapazonunda **Setir** adlı sətrə neçə dəfə kəsişmədən iştirak etdiyini qaytarır. (0 və susma halında sətrin uzunluğu) >>> Setir="abcdeabcabcdeabcd" >>> Setir.count("abc", 0,55) 4 >>> Setir.count("zzz", 0,55) 0
Setir.expandtabs([tabsize])	Sətrin surətini qaytarır. Bu halda cari sütundan asılı olaraq bütün tabulyasiya simvolları ya bir simvolla və ya probellə əvəz edilir.

Funksiya və metodlar	Təyinatı
	Əgər**tabsize**göstərilməmişsə, onda 8 probel nəzərdə tutulur. >>> Setir="Aaa bb ccccc dddd" >>> Setir.expandtabs(4) 'Aaa bb ccccc dddd'
Setir.lstrip([chars])	Sətrin əvvəlində probel kimi istifadə edilən simvolların silinməsi >>> Setir="==========abcdefgh========" >>> Setir.lstrip("=") 'abcdefgh========'
Setir.rstrip([chars])	Sətrin sonunda probel kimi istifadə edilən simvolların silinməsi >>> Setir="==========abcdefgh========" >>> Setir.rstrip("=") '==========abcdefgh'
Setir.strip([chars])	Sətrin əvvəlində və sonunda probel kimi istifadə edilən simvolların silinməsi >>> Setir="==========abcdefgh========" >>> Setir.strip("=") 'abcdefgh'
Setir.partition(şablon)	Verilmiş şablona görə mətni üç hissəyə bölür: şablona qədərki hissə, şablon və şablondan sonrakı hissə. Qiyməti həmin hissələrdən ibarət olan kortej yaradır. Əgər şablon olmasa, birinci elementi həmin mətn, sonrakı elementləri boş olan kortej yaradır. >>> Setir="aaaaaa==bbbbbb" >>> Setir.partition("==") ('aaaaaa', '==', 'bbbbbb') >>> Setir.partition("-") ('aaaaaa==bbbbbb', '', '')
Setir.rpartition(sep)	Verilmiş şablona görə mətni üç hissəyə bölür: sonuncu şablona qədərki hissə, şablon və sonuncu şablondan sonrakı hissə. Qiyməti həmin hissələrdən ibarət olan kortej yaradır. Əgər şablon olmasa, əvvəlki elementləri boş, üçüncü elementi isə həmin mətn olan kortej yaradır. >>> Setir="aaaaaa==bbbbbb==cccccc" >>> Setir.rpartition("==") ('aaaaaa==bbbbbb', '==', 'cccccc')

Funksiya və metodlar	Təyinatı
Setir.swapcase()	Alt reqistrdə olan hərfləri üst reqistrə, üst reqistrdə olan hərfləri isə alt reqistrə çevirir. >>> Setir="ABCDEFGHabcdefgh" >>> Setir.swapcase() 'abcdefghABCDEFGH'
Setir.title()	Mətndəki hər sözün birinci hərfini üst reqistrə, digərlərini isə alt reqistrə çevirir. >>> Setir="Bu biR yoxlama mətniDir." >>> Setir.title() 'Bu Bir Yoxlama Mətnidir.'
Setir.zfill(width)	Sətrin uzunluğunu, ehtiyac olsa əvvəlinə sıfırlar artırmaqla, verilmiş **width** uzunluğuna qədər böyüdür. >>> Setir="aaaaabbbbbcccccdddddeeeeefffff" >>> Setir.zfill(40) '0000000000aaaaabbbbbcccccdddddeeeeefffff' >>> Setir.zfill(20) 'aaaaabbbbbcccccdddddeeeeefffff'
Setir.ljust(width, fillchar)	Sətrin uzunluğunu, ehtiyac olsa sonuna **fillchar** simvol artırmaqla, verilmiş **width** uzunluğuna qədər böyüdür. >>> Setir="aaaaabbbbbcccccdddddeeeeefffff" >>> Setir.ljust(40, "=") 'aaaaabbbbbcccccdddddeeeeefffff==========' >>> Setir.ljust(40, "+") 'aaaaabbbbbcccccdddddeeeeefffff++++++++++'
Setir.rjust(width, fillchar=" ")	Sətrin uzunluğunu, ehtiyac olsa əvvəlinə **fillchar** simvol artırmaqla, verilmiş **width** uzunluğuna qədər böyüdür. >>> Setir="aaaaabbbbbcccccdddddeeeeefffff" >>> Setir.rjust(40, "+") '++++++++++aaaaabbbbbcccccdddddeeeeefffff'
Setir.format(*args, **kwargs)	Sətrin formatlanması.

Sətrin formatlanması. Format metodu

Tez-tez proqramın icrası nəticəsində alınan bəzi verilənləri əlavə etməklə yeni sətir yaratmağa ehtiyac duyulur. Belə işi sətirlərin formatlanması vasitəsilə görmək olar. Formatlanmağı **%** operatoru və **format** metodunun köməyilə aparmaq olar.

format metodu ilə sətirlərin formatlanması

format metodu vasitəsilə yalnız bir arqumentdən istifadə ediləcəksə, onda qiymət arqumentin özü olacaq:

```
>>> 'Salam, {}!'.format('Abdulla')
'Salam, Abdulla!'
>>> Ad="Abdulla"
>>> 'Salam, {}!'.format(Ad)
'Salam, Abdulla!'
```

Misala diqqətlə baxsanız görərsiniz ki, eyni əməliyyatı biz iki variantda yerinə yetirdik. Birinci variantda adı sətir kimi verdik. İkinci variantda isə mənimsətmə yolu ilə **Ad** dəyişəni yaratdıq. Sonra həmin dəyişəndən **format** metodunda istifadə etdik. Əlbəttə ki, real proqram yazanda ikinci variantdan daha çox istifadə ediləcək.

Bir neçə arqumentdən istifadə edən zaman arqument kimi əvəzetmə sətri (adi və ya adlanmış) olacaq. Aşağıdakı misalda bu göstərilib:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>> 'Koordinatlar: {eni}, {uzunu}'.format(latitude='37.24N', longitude='-115.81W')
' Koordinatlar: 37.24N, -115.81W'
>>> coord = {'eni: '37.24N', 'uzunu': '-115.81W'}
>>> ' Koordinatlar: {latitude}, {longitude}'.format(**coord)
' Koordinatlar: 37.24N, -115.81W'
```

format metodunun imkanları daha çoxdur. Onun sintaksisi belədir:

```
əvəzetmə sahəsi::="{"[sahənin adı]["!"çevirmə][":"spesifikasiya]"}"

sahənin adı::=arg_name("."atributun adı|"["indeks"]")*
```

```
çevirmə::="r"(daxili təmsil)|"s"(insan üçün təmsili)

spesifikasiya::=aşağı bax:
```

Məsələn:

```
>>> "Units destroyed: {players[0]}".format(players = [1, 2, 3])
'Units destroyed: 1'
>>> "Units destroyed: {players[0]!r}".format(players = ['1', '2', '3'])
```

Formatın spesifikasiyası aşağıdakı kimidir:

```
spesifikasiya ::=  [[fill]align][sign][#][0][width][,][.precision][type]
doldurucu::=  '{' və ya '}' – dan savayı simvol
nizamlama                ::=  "<" | ">" | "=" | "^"
işarə       ::=  "+" | "-" | " "
en          ::=  integer
dəqiqlik  ::=  integer
 tip::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |"n" | "o" | "s" | "x" | "X" | "%"
```

Nizamlama doldurucu simvol vasitəsilə həyata keçirilir. Nizamlamanın aşağıdakı variantları var:

Bayraq	Qiyməti
'<'	Sağ tərəfə görə nizamlama (susma halı).
'>'	Obyekti sol tərəfə görə nizamlayır
'='	Doldurucu yalnız ədədlərdən əvvəl işarədən sonra olacaq. Yalnız ədədlərin tipləri ilə işləyir.
'^'	Mərkəzə görə nizamlayır.

“İşarə” opsiyası yalnız ədədlər üçün istifadə edilir və aşağıdakı qiymətləri alır:

Bayraq	Qiymət
'+'	İşarə bütün ədədlər üçün istifadə edilməlidir.
'-'	'-' mənfi ədədlər üçün, müsbət ədədlər üçün heç nə.
'Probel'	'-' mənfi ədədlər üçün, müsbət ədədlər üçün probel.

"Tip" sahəsi aşağıdakı qiymətləri ala bilər:

Tip	Qiymət
'd', 'i', 'u'	Onluq say sistemində ədəd.
'o'	Səkkizlik say sistemində ədəd.
'x'	Onaltılıq say sistemində ədəd (hərflər alt reqistrdədir).
'X'	Onaltılıq say sistemində ədəd (hərflər üst reqistrdədir).
'e'	Eksponentli sürüşkən vergüllü ədəd (eksponent alt reqistrdədir).
'E'	Eksponentli sürüşkən vergüllü ədəd (eksponent üst reqistrdədir).
'f', 'F'	Sürüşkən vergüllü ədəd (adi format).
'g'	Sürüşkən vergüllü ədəd. Eksponent **-4** –dən kiçik və bərabərdirsə eksponentli (eksponent alt reqistrdə), əks halda adi format.
'G'	Sürüşkən vergüllü ədəd. Eksponent **-4** –dən kiçik və bərabərdirsə eksponentli (eksponent üst reqistrdə), əks halda adi format.
'c'	Simvol (bir simvoldan ibarət sətir və simvolun kodu olan ədəd).
's'	Sətir.
'%'	Ədəd **100**-ə vurulur, sürüşkən vergüllü ədəd göstərilir və ondan sonra **%** simvolu gəlir.

İndekslər

Python dilində indeksə görə iş mövcuddur. Sətrin daxilində olan simvolların (siyahının və kortejlərin elementlərinin) sıra nömrəsi aşağıdakı sxemə əsasən müəyyən edilir:

İstiqamət	Sıralama (n simvolu olan sətir)								
Soldan sağa və ya əvvəldən sona	**0**	**1**	**2**	**3**	**4**	**...**	**n-3**	**n-2**	**n-1**
Sağdan sola və ya sondan əvvələ	**-n**	**-n+1**	**-n+2**	**-n+3**	**...**	**-4**	**-3**	**-2**	**-1**

Məsələn: "**Qarabağ**" sətri üçün nömrələmə belə olacaq:

Q	**a**	**r**	**a**	**b**	**a**	**ğ**
0	**1**	**2**	**3**	**4**	**5**	**6**
-7	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**

Məsələn belə bir misala baxaq:

```
>>> a = [1, 3, 8, 7]
>>> a[0]
1
>>> a[3]
7
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Sonuncu yazılış səhv verir. **a** siyahısı dörd elementdən ibarətdir. İndeksləmə 0-la 3 arasında ola bilər. Ona görə də 4 nömrəli indeks mövcud deyil (**IndexError: list index out of range**).

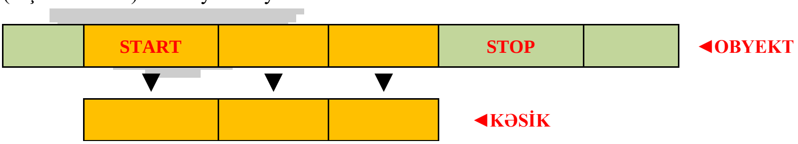
Yuxarıdakı misalda nümunə kimi siyahı götürülmüşdü. Ancaq nümunə kimi siyahı və kortej də götürülə bilər.

Python dili mənfi indeksləməni də götürür. Bu halda sıralama sondan başlayır. Məsələn:

```
>>> a = [1, 3, 8, 7]
>>> a[-1]
7
>>> a[-4]
1
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Kəsiklər (slices)

Python dilində indeksdən başqa kəsiklər də mövcuddur. **Kəsik (slice)** — verilmiş obyektdən bir simvolun (elementin), sətrin və ya altardıcıllığnı müəyyən fraqmentinin ayrılmasından (seçilməsindən) alınan yeni obyektdir.



item [START:STOP:STEP]

START – kəsiyin başlanğıc nömrəsi (indeksi),

STOP – kəsiyin son sərhəddi-indeksi (həmi nömrəli element kəsiyə daxil olmur),

STEP – addım.

Susma halında START = 0, STOP = obyektin uzunluğu, STEP = 1. İstifadə edilən zaman hansısa (ola bilər ki, hamısı) parametr buraxıla bilər.

```
>>> vektor = [2, 3, 8, 4]
>>> vektor [:]
[2, 3, 8, 4]
>>> vektor [1:]
[3, 8, 4]
>>> vektor [:3]
[2, 3, 8]
>>> vektor [::2]
[2, 8]
```

İndeks bölməsindən bildiyimiz kimi mənfi sıralamadan da istifadə edilir. Kəsiklər üçün də bu keçərlidir. Bütün parametrlər mənfi ola bilər.

```
>>> vektor = [2, 3, 8, 4]
>>> vektor [::-1]
[4, 8, 3, 2]
>>> vektor [:-2]
[2, 3]
>>> vektor [-2::-1]
[8, 3, 2]
>>> vektor [1:4:-1]
[]
```

Sonuncu proqram kodunda boş siyahı alınıb. Çünki, START < STOP, a STEP isə mənfidir. Qiymətlər diapazonu obyektdən (daha doğrusu indekslər elementlərin sayından) kənara çıxdıqda da boş siyahı alınacaq:

```
>>> vektor = [2, 3, 8, 4]
>>> vektor[30:35]
[]
```

Kəsik vasitəsilə obyektdən element çıxarmaq, həm də eəlement əlavə etmək və silmək olar (kortejlər üçün yox).

```
>>> vektor = [2, 3, 8, 4]
>>> vektor [1:3] = [0, 0, 0]
>>> vektor
[2, 0, 0, 0, 4]
>>> del vektor [:-3]
>>> vektor
[0, 0, 4]
```

Yuxarıdakı nümunələrdə kəsiklər siyahılar əsasında verilib. Eyni şeyi sətirlər üçün də etmək olar.

Tutaq ki, Metn = “Salam” sətri verilib. Aşağıdakı misalda kəsiyin sətrə tətbiqi göstərilib.

```
>>> Metn = "Salam"
>>> Metn [1]
'a'
```

```
>>> Metn [3]
'a'
>>> Metn [-1]
'm'
>>> Metn [4]
'm'
>>> Metn [1:3]
'al'
>>> Metn [1:]
'alam'
>>> Metn [1:50]
'alam'
>>> Metn [:-1]
'Sala'
>>> Metn [:4]
'Sala'
>>> Metn [1:5:2]
'aa'
>>> Metn [0:5:2]
'Slm'
```

>>>**Metn [1:50]** əmrində **50** sətrin uzunluğundan böyük olsa da (**50** > **len(Metn)=5**), əməliyyat sətrin uzunluğuna bərabər say qədər yerinə yetiriləcək.

Əgər kəsiyin başlanğıc indeksi sətrin uzunluğundan böyük və bərabər olsa, onda interpretaror ***IndexError: string index out of range*** məzmunlu səhv verəcək.

Misallardan göründüyü kimi **Metn[a:b]** kəsiyi bizə **a** simvolundan başlayaraq **b** simvolu daxil olmamaqla ona qədər **(b-a)** sayda simvolu olan altsətir verəcək.

Eyni bir elementi (simvolu) həm müsbət, həm də mənfi indekslə göstərmək olar.

Metn [4] = Metn [-1] = 'm'

Verilənlər strukturu

Python dilində verilənlər strukturu deyəndə hər hansı veriləni yadda saxlamaq üçün istifadə edilən struktur nəzərdə tutulur. Başqa sözə o o rabitəli verilənləri yadda saxlamaq üçün istifadə edilir. Python dilində 4 növ daxili verilənlər strukturu mövcuddur:

- **siyahı,**
- **kortej,**
- **lüğət,**
- **çoxluq.**

Siyahılar (list). Siyahıların funksiyaları və metodları

Python dilində digər proqramlaşdırma dillərindəki kimi adi başa düşdüyümüz massiv anlayışı yoxdur. Bunun əvəzində siyahılardan (**list**) istifadə edilir.

Python dilində **siyahı** nizamlanmış obyektlər toplusudur. Massivdən fərqli olaraq siyahılarda obyektlər müxtəlif tipli (ədəd, sətir, siyahı və kortej də daxil olmaqla digər strukturlar) ola bilərlər. Siyahıya daxil olan obyektlər siyahının elementləri adlanır.

Siyahıları "dinamik massiv" anlayışınıngenişlənməsi kimi də başa düşmək olar. Biz siyahının həm hər bir elementi ilə də, həmdə bütün siyahı ilə işləyə bilərik. Məsələn, siyahıya element artırmaq və siyahıdan element silmək, siyahının bir hissəsini köçürmək, nizamlamaq və s.

Siyahılara nümunələr:

```
[12, 14, 21, 22, -55, 99]  # tam ədədlərdən ibarət siyahı

[12.34, 23.45, 323.32, 21.43, 23.57]  # kəsr ədədlərdən ibarət siyahı

[“Aysu”, “Ömər”, “Araz”, “Çiçək”, “Arif”]  # sətirlərdən ibarət siyahı

[“Bakı”, “İstanbul”, 23, 31]  # qarışıq siyahı

[ [0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 1] ]  # siyahılardan ibarət siyahı
```

Sonuncu sətir siyahılardan massiv kimi istifadə etməyə bir bir nümunədir.

Siyahıdan istifadə etmək üçün onu yaratmaq lazımdır. Bunun üçün bir neçə üsul var. Python dilində siyahının yaradılmasının ən sadə yolu belədir: istənilən obyekti, məsələn sətri, daxili **list** funksiyası vasitəsilə siyahıya çeviririk:

```
>>> list('siyahı')
['s', 'i', 'y', 'a', 'h', 'ı']
```

Siyahını literallar vasitəsilə də yaratmaq olar. Bu üsulda siyahının elementlərini kvadrat mötərizədə sadalamaq lazımdır.

```
>>>bosh = []  # Boş siyahı
>>>dolu = ['s', 'i', ['yahı'], 3]
>>>bosh
[]
>>>dolu
['s', 'i', ['yahı'], 3]
```

Misaldan göründüyü kimi, siyahı ixtiyarı sayda ixtiyari obyektlərdən (o cümlədən iç-içə) ibarət və ya boş ola bilər.

Siyahılarda elementlərin nömrələnməsi sıfırdan (**0**) başlayır.

Digər proqramlaşdırma dillərində massivə eyni qiymətlər mənimsədəndə dövr operatorundan istifadə edilir. Python dilində isə bu qısa formada yazılır. Məsələn, 5 elementi olan **B** adlı siyahı yaradıb, onun bütün elementlərinə iki (**2**) qiyməti mənimsətmək istəyiriksə, aşağıdakı yazılışdıan istifadə edə bilərik.

```
>>>B = [2] * 5
>>>B
[2, 2, 2, 2, 2]
```

Birinci sətirdə [2] yazılışı qiyməti 2-yə bərabər olan birölçülü siyahı deməkdir. Beşə (**5**) ***vurma*** isə onu bildirir ki, beş ədəd eyni siyahı vahid siyahıda birləşir.

Siyahıdakı elementlərin sayını ədədlə yox, dəyişən kimi də göstərmək olar. Bunun üçün dəyişənə öncə qiymət vermək lazımdır. Məsələn:

```
>>> N=5
>>> B=[2]*N
>>> B
[2, 2, 2, 2, 2]
```

Siyahı generatoru. İfadəni ardıcıllığın hər bir elementinə tətbiq etməklə yeni siyahı yaratmaq siyahı generatoru adlanır.Tutaq ki, siyahının elementlərinin hamısına elementin sıra sayının (indeksinin) kavadratını mənimsətmək lazımdır. Bu halda yazılış belə olacaq.

```
>>> B = [i * i for i in range(5)]
>>> B
[0, 1, 4, 9, 16]
```

Bu ***siyahı generatorudur***. Biz birinci sətirdə **for i in range(5)** dövrünü görürük. Bu dövr bütün 0-dan 4-dək (5 ədəd) olan tam ədədləri seçir və onların kvadratını (***i*i***) hesablayır və alınan nəticələr əsasında siyahı yaradır (kvadrat mötərizə bunu göstərir).

Generatora aid ikinci misal:

```
>>> c = [c * 3 for c in 'alma']
>>> c
['aaa', 'lll', 'mmm', 'aaa']
```

Generatora aid daha çətin konstruksiyalı üçüncü misal:

```
>>> c = [c * 3 for c in 'armud' if c != 'r']
>>> c
['aaa', 'mmm', 'uuu', 'ddd']
>>> c = [c + d for c in 'armud' if c != 'm' for d in 'alma' if d != 'a']
>>> c
['al', 'am', 'rl', 'rm', 'ul', 'um', 'dl', 'dm']
```

Əgər özünüzə inanmırsınızsa, çətin konstruksiya yaradan generatorlardan istifadə etməyin!

Daxili funksiyalar vasitəsilə siyahının minimumunu və maksimumu tapmaq olar:

```
>>> minimum = min(B)
>>> maksimum = max(B)
>>> minimum
0
>>> maksimum
16
```

Siyahının elementlərini nizamlamaq da olar. Bunu daxili **sort()** funksiyası ilə etmək olar:

```
>>> B.sort()
>>> B
[0, 1, 4, 9, 16]
>>> B=B[::-1]
>>> B
[16, 9, 4, 1, 0]
```

Python dilində siyahının ***reversi*** (elementlərin tərsinə düzülüşü) çox sadəcə B=B[::-1] sətri ilə həyata keçirilir. Biz yuxarıda baza əməliyyatlar sırasında bu əmri (kəsiyin alınması) göstərmişdik.

Bəzən siyahının bir siyahının müəyyən şərtləri ödəyən (məsələn, bütün müsbət) elementlərini seçib digər bir siyahıya yığmaq lazımdır. Python dilində bu çox sadəcə, bir generator vasitəsilə edilir:

```
>>> Baza = [4, 5, -6, 8, -3, 4, 3, -3]
>>> Baza
[4, 5, -6, 8, -3, 4, 3, -3]
>>> Son = [x for x in Baza if x > 0]
>>> Son
[4, 5, 8, 4, 3]
```

Daha asan başa düşmək üçün əmri hissələrə bölmək olar:

```
Son = [x
for x in Baza
if x > 0]
```

Şərtə diqqət eləsək görərik ki, biz **Baza** siyahısının bütün elementlərini saf-çürük edirik (**for x in Baza** dövrü), yalnız müsbət ədədləri saxlayırıq (**if x > 0**) və sonda yeni siyahı qururuq ([x ...]).

Şərti dəyişək. İndi biz **Son** siyahısına bütün təkrar olunmayan elementləri yığmaq istəyirik. Python dilində bunu bir operatorla etmək olur. Əvvəlki misalda **Baza** siyahısı üçün proqram kodu belə olar:

```
>>> Son = list ( set(Baza) )
>>> Son
[3, 4, 5, 8, -6, -3]
```

Əvvəlcə **Baza** siyahısı əsasında bütün təkrar olunan elementlər silinməklə çoxluq yaradılır (**set**). Sonra çoxluq yenidən siyahıya çevrilir (**list**).
Siyahılarla iş zamanı yadda saxlamaq lazımdır ki, *dəyişən-siyahı* özü *müraciətdir*. Misala baxaq:

```
>>> A = [1, 2, 3]
>>> B = A
>>> A
[1, 2, 3]
>>> B
[1, 2, 3]
```

Bu misalda yaddaşda bir siyahı yaradılır. Amma bu siyahıya **A** və **B** adları ilə 2 müraciət var. **A** siyahısını dəyişdikdə **B** siyahısı da avtomatik dəyişəcək.
Əgər siyahının surətini yaratmaq istəyiriksə, onda bütün elementlərin kəsiyini götürmək lazımdır:

```
>>> B = A[:]
>>> A
[1, 2, 3]
>>> B
[1, 2, 3]
```

Nəticədə ikinci asılı olmayan siyahı yaradılacaq. Bu halda **A** siyahısının dəyişməsi **B** siyahısına təsir etməyəcək:
A [1, 2, 3]
B [1, 2, 3]

Python dilində siyahının elementləri kimi dəyişənlərdən də istifadə edilə bilər. Məsələn, öndə misalda verilmiş korteji başqa cür də qurmaq olar:

```
>>> gün=21
>>> ay="dekabr"
>>> il=1955
>>> dogum_tarixi=[gün, ay, il]
>>> dogum_tarixi
[21, 'dekabr', 1955]
>>> dogum_tarixi[0]
21
>>> gün
21
>>> ay="yanvar"
>>> ay
'yanvar'
```

```
>>> dogum_tarixi
[21, 'dekabr', 1955]
>>> dogum_tarixi[1] = "yanvar"
>>> dogum_tarixi
[21, 'yanvar', 1955]
```

Bu misalda öncə **gün, ay, il** adlı dəyişənlər yaradılır. Daha sonra bu dəyişənlərdən element kimi istifadə edilməklə **doğum_tarixi** adlı siyahı yaradılır. Bu halda siyahının **dogum_tarixi[0]** elementi ilə **gün** dəyişəni eyni bir qiymətə istiqamətləniblər. Siyahının digər elementləri haqqında da eyni fikri demək olar. Burada bir şeyə də diqqət etmək lazımdır. Siyahının yaranmasında iştirak edən **ay** dəyişəninin qiymətinin dəyişməsi siyahının dəyişməsinə təsir göstərmir. Çünki, siyahı yaranandan sonra onlar arasındakı bağlılıq itir. Amma siyahının elementinin qiyməti dəyişir.

dogum_tarixi[1] = "yanvar" kod sətri **dogum_tarixi** adlı siyahının bir saylı elementinin qiymətini dəyişir. Misalda bu dəyişiklik öz əksini tapıb. →**[21, 'yanvar', 1955]**

Siyahıların funksiyaları və metodları (üsulları)

Funksiya və metodlar	Təyinatı
Siyahi.append(x)	Siyahının sonuna element əlavə edir. >>> Siyahi=[12, 14, 21, 22, -55, 99] >>> Siyahi.append(101) >>> Siyahi [12, 14, 21, 22, -55, 99, 101] >>> Siyahi.append(8) >>> Siyahi [12, 14, 21, 22, -55, 99, 101, 8]
Siyahi.extend(L)	**L** siyahısının elementlərini **Siyahi** adlı siyahını sonuna artırmaqla genişləndirir. >>> Siyahi=[12, 14, 21, 22, -55, 99, 101, 8] >>> L=[5,6,9] >>> Siyahi.extend(L) >>> Siyahi [12, 14, 21, 22, -55, 99, 101, 8, 5, 6, 9]
Siyahi.insert(i, x)	Siyahıda ***i***-ci elemet kimi **x** qiymətini əlavə edir. >>> Siyahi=[12, 14, 21, 22, -55, 99, 101] >>> Siyahi.insert(3, 23) >>> Siyahi [12, 14, 21, 23, 22, -55, 99, 101]
Siyahi.remove(x)	Siyahıda **x** qiyməti olan birinci elementi silir. >>> Siyahi=[12, 14, 21, 22, -55, 22, 22, 99, 101] >>> Siyahi.remove(22) >>> Siyahi [12, 14, 21, -55, 22, 22, 99, 101]
Siyahi.pop([i])	Siyahıdakı ***i***-ci elementi silir və onun qiymətini cavab kimi qaytarır. Əgər indeks göstərilməyibsə, sonuncu element silinir. >>> Siyahi=[12, 14, 21, 22, -55, 99, 101] >>> Siyahi.pop(2) 21 >>> Siyahi [12, 14, 22, -55, 99, 101]

Funksiya və metodlar	Təyinatı
Siyahi.index(x, [start [,end]])	**Start** və **end** arasında yerləşən və qiymәti **x** olan birinci elementin indeksini qaytarır. >>> Siyahi=[12, 22, 14, 21, 22, -55, 22, 99, 101] >>> Siyahi.index(22,3,8) 4
Siyahi.count(x)	Siyahıdakı **x** qiymətli elementlərin sayını verir. >>> Siyahi=[12, 22, 14, 21, 22, -55, 22, 99, 101] >>> Siyahi.count(22) 3
Siyahi.sort([key = funksiya])	Siyahını funksiyaya əsasən nizamlayır. >>> Siyahi=[-55, 12, 14, 21, 22, 22, 22, 99, 101] >>> Siyahi.sort(key = abs) >>> Siyahi [12, 14, 21, 22, 22, 22, -55, 99, 101]
Siyahi.reverse()	Siyahını tərsinə çevirir. >>> Siyahi=[-55, 12, 14, 21, 22, 22, 22, 99, 101] >>> Siyahi.reverse() >>> Siyahi [101, 99, 22, 22, 22, 21, 14, 12, -55]
Siyahi.copy()	Siyahının yeni surətini alır. >>> Siyahi=[-55, 12, 14, 21, 22, 22, 22, 99, 101] >>> L=Siyahi.copy() >>> L [-55, 12, 14, 21, 22, 22, 22, 99, 101]
Siyahi.clear()	Siyahının elementlərini silir. >>> Siyahi=[-55, 12, 14, 21, 22, 22, 22, 99, 101] >>> Siyahi.clear() >>> Siyahi []

Burada bir şeyi qeyd etmək lazımdır ki, sətirlər üçün metodlardan fərqli olaraq siyahıların metodları siyahının özündə düzəliş aparır. Ona görə də icranın nəticəsini yenidən siyahının elementinə mənimsətməyə ehtiyac yoxdur.

```
>>> Siyahi=[-55, 12, 14, 21, 22, 22, 22, 99, 101]
>>> print(Siyahi.count(22), Siyahi.count(99), Siyahi.count(88))
3 1 0
```

Gördüyünüz misalın cavabını araşdırın.

Kortejlər (tuple)

Kortej (tuple) siyahının xüsusi bir halıdır. Kortej siyahıdan onunla fərlənir ki, onu dəyişmək olmur, yəni kortej dəyişilməyən siyahıdır. Kvadrat mötərizə ilə yazılan siyahıdan fərqli olaraq kortej adi yumru mötərizə ilə yazılır. Kortejin elementləri müxtəlif tip obyekt (ədəd, sətir və kortej də olmaqla digər struktur) ola bilər. Elementlər bir-biri ilə vergüllə ayrılır.

Məsələn: **number = (1, 2, 3, 4, 5)**.

Əgər siyahı varsa, kortejə nə ehtiyac var?

- Təsadüfdən qorunmaq. Çox vaxt proqramlar dəyişməz siyahılardan istifadə edir (məsələn, ayların adı). Əgər bu məlumat yaddaşda **siyahı (list)** kimi saxlanılsa, məqsədli və ya təsadüfən dəyişdirilə bilər. Siyahıdan fərqli olaraq kortejə yeni element əlavə etmək, elementini dəyişmək və silmək olmaz. Amma yeni kortej yaradıb onu həmin dəyişənlə əlaqələndirmək olar.
- Kiçik ölçülü olması.
- Kortejlərdən lüğətlərin açarı kimi istifadə etmək.

```
>>> d = {(1, 1, 1) : 1}
>>> d
{(1, 1, 1): 1}
>>> d = {[1, 1, 1] : 1}
Traceback (most recent call last):
File "", line 1, in
d = {[1, 1, 1] : 1}
TypeError: unhashable type: 'list'
```

Kortejlərin üstünlüyünü bildik. İndi isə necə işləməyi öyrənək. Kortejlərlə işləmək daha çox siyahılarla işləməyə oxşayır.

Kortej boş ola bilər. Əvvəlcə boş kortej yaradaq:

```
>>>bosh = tuple() # tuple() daxili funksiyası vasitəsilə
>>> bosh
```

```
()
>>> bosh = () # Kortejin literalı vasitəsilə
>>> bosh
()
```

Bir elementdən ibarət kortej yaradaq:

```
>>> kort = ('aa')
>>> kort
'aa'
```

Misaldan göründüyü kimi, biz kortej almaq əvəzinə sətir aldıq. Bəs korteji necə ala bilərik? Sadəcə olaraq korteje bir element əlavə etmək üçün daxil edilən elementdən sonra **vergül** (**“,”**) yazmaq lazımdır.

```
>>> kort = ('aa', )
>>> kort
('aa',)
```

Əla, alındı. Sadəcə bir vergül məzmuna böyük təsir göstərə bilir. **'s'** yazısı onun sətir olmasını, **('s',)** yazısı isə kortej olmasını bildirir. Beləliklə, bir elementdən ibarət kortej yaratmaq üçün həmin elementin qiymətindən sonra mütləq vergül işarəsi qoymaq lazımdır.

Korteji belə də yaratmaq olar:

```
>>> kort = 'aa',
>>> kort
('aa',)
```

Amma yenə də, gözlənilməzliklər olmasın deyə mötərizələrin yazılması məsləhətdir.

Yuxarıda dediyimiz kimi kortejin elementləri müxtəlif tipli də ola bilər:

dogum_tarixi=(21, “dekabr”, 1955).

Python dilində kortejin elementləri kimi dəyişənlərdən də istifadə edilə bilər. Məsələn, öndə misalda verilmiş korteji başqa cür də qurmaq olar:

```
>>> gün=21
>>> ay="dekabr"
>>> il=1955
>>> dogum_tarixi=(gün, ay, il)
>>> dogum_tarixi
(21, 'dekabr', 1955)
>>> dogum_tarixi[0]
21
>>> gün
```

```
21
```

Bu misalda öncə **gün, ay, il** adlı dəyişənlər yaradılır. Daha sonra bu dəyişənlərdən element kimi istifadə edilməklə **doğum_tarixi** adlı kortej yaradılır. Bu halda kortejin **dogum_tarixi[0]** elementi ilə **gün** dəyişəni eyni bir qiymətə istiqamətləniblər. Kortejin digər elementləri haqqında da eyni fikri demək olar.

Bildiyimiz kimi kortej siyahıya oxşasa da, onun elementlərinin qiymətini dəyişmək olmaz. Məsələn, əvvəlki misalda proqram kodunu davam etdirsək dediklərimizi təsdiq edə bilərik.

```
>>> ay="yanvar"
>>> ay
'yanvar'
>>> dogum_tarixi
(21, 'dekabr', 1955)
>>> dogum_tarixi[1] = "yanvar"
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    dogum_tarixi[1] = "yanvar"
TypeError: 'tuple' object does not support item assignment
```

Misaldan göründüyü kimi **ay** dəyişəninin qiymətini dəyişə bilərik. Amma bu dəyişiklik kortejin **dogum_tarixi[1]** elementinin dəyişməsinə səbəb olmur.

dogum_tarixi[1] = "yanvar" elementinin birbaşa dəyişilməsi isə səhv kimi dəyərləndirilir. Dediklərimiz təsdiq olundu. Bir dəfə yaranan kortejin qiymətini heç bir yolla dəyişmək olmaz.

Python dilində müəyyən funksiyaların yerinə yetirilməsi nəticəsində də kortej alına bilər. Məsələn, **divmod()** funksiyası iki elementdən ibarət kortej yaradır.
Hansısa əşyalar, varlıqlar və ya hadisələr müəyyən sonlu sayda xassələri ilə xarakterizə olunurlarsa, bu halda kortejlərdən istifadə etmək məqsədəuyğundur. Məsələn, avtomobilin markasını və onun göstəricilərini, ya da şagirdin adını və onun yarımillik (illik) qiymətlərini kortej kimi yadda saxlamaq olar.
yigma = (11, 12, 13)və ya**yigma = 11, 12, 13**–korteji yaradır (yığıb bükür).

a1, a2, a3 = yıgma–korteji açır.

Öndə dediyimiz kimi korteji dəyişmək olmaz. Bunu etmək üçün öncə korteji açmalı, daha sonra dəyişənlər üzərində əməliyyat aparıb sonda yenidən bükməliyik.

```
>>> yigma = (11, 12, 13)
>>> yigma
(11, 12, 13)
>>> a1, a2, a3 = yigma
>>> a1, a2, a3
>>> a1
11
>>> a2
12
>>> a3
13
>>> a1=2*a1
```

```
>>> a2=2*a2
>>> a3=2*a3
>>> a1, a2, a3
(22, 24, 26)
>>> yigma
(11, 12, 13)
>>> yigma = (a1, a2, a3)
>>> yigma
(22, 24, 26)
```

Kortejlər siyahılar kimi müqayisə olunur.

Sətir tipli obyektdən kortej yaratmaq üçün **tuple()** funksiyasından istifadə etmək olar.

```
>> kort = tuple('Salam, Bakı!')
>>> kort
('S', 'a', 'l', 'a', 'm', ',', ' ', 'B', 'a', 'k', 'ı', '!')
```

Siyahılar üzərində aparılan və siyahını dəyişməyən bütün bütün əmrlər (cəm, ədədə vurma, index() və count() və bəzi digər əmrlər) kortejlər üçün də tətbiq edilir. Bundan başqa elementlərin yerini müxtəlif formada dəyişmək olar və s.

Məsələn, proqramlaşdırmada ən çox tətbiq edilən iki dəyişənin qiymətini dəyişməyə aid nümunə Python dilində çox sadə yerinə yetirilir:

```
>>> a=5
>>> b=6
>>> a,b
(5, 6)
>>> a,b=b,a
>>> a,b
(6, 5)
```

Əgər koordinat sistemi ilə bağlı məsələlərhəll ediriksə, onda hər bir nöqtə haqqında koordinat göstəricilərini kortej formasında saxlamaq məqsədəuyğundur. İkiölçülü koordinat üçün bunu **(x,y)** cütü şəklində vermək olar. Məsələn sınıq xətti qurmaq üçün istifadə edilən nöqtələrin koordinatlarını kortejin elementləri formasında verə bilərik:

L = [(0,0), (0,2), (5,-5), (6, 6)]

Lüğətlər (dict) vəonlarla iş. Lüğətlərin metodları (üsulları)

Python dilində ən geniş istifadə edilən və ən çətin verilənlər tipindən (sətir və siyahılarla yanaşı) biri də lüğətdir. **Lüğət** siyahı kimi dəyişən və sətirlə siyahı kimi nizamlanmamış ***"açar:qiymət"*** cütlüyüdür. Onları bəzən assosiativ massivlər və ya xeş-cədvəllər adlandırırlar.

Sətirlər, siyahılar və kortejlərdə tam ədədlərdən indeks kimi istifadə edilib obyektin elementinə indeksə görə müraciət edilirdi.

Lüğətlər isə tamam başqa formada qurulurlar. Lüğətlər siyahılar kimi ən sadə, çevik və güclü toplu tipidirlər. Lüğətlərdə ayrı-ayrı elementlərə müraciət etmək üçün verilənlər bazasında olduğu kimi açarlardan (daha doğrusu açar indekslərdən) istifadə edilir. Lüğətlərin dəyişməz açarları olur və onlar məhdudsuz sayda arta bilirlər. İxtiyari dəyişilməyən obyekt (sətir, ədəd, sabitlərdən ibarət siyahı) indeks ola bilər. Proqram mətnində lüğətlər daxilində lüğətin elementləri olan fiqurlu mötərizə **{}** ilə göstərilir. Lüğətin hər bir elementinə **"indeks:qiymət"** formatlı indeks uyğun gəlməlidir. Lüğətin elementlərinə uyğun indekslər vasitəsilə müraciət edilir. Mövcud olmayan indeksə müraciət etdikdə səhv verir.

Siyahılar nizamlanmış olsa da, lüğətlər nizamlanmamışdır. Lüğətlərin əsas xüsusiyyətləri bunlardır:

- Lüğətin elementinə müraciət indeksə görə yox, açara görə aparılır. Siyahılara oxşar olaraq dövrdə lüğətin elementlərinə müraciət açar vasitəsilə aparılır.
- Lüğətin məlumatları nizamlanmamış şəkildə yadda saxlanılır. Açarlar özü də onlar daxil olunduqları ardıcıllıqla saxlanmaya bilərlər.
- Siyahılarda olduğu kimi lüğət özündə daxili lüğəti saxlaya bilər. Lüğətin obyektləri ədəd, siyahı, kortej tipli ola bilər.
- Lüğətlər sürətli axtarışlı keş-cədvəllər kimi realizə ediliblər.
- Lüğətlər siyahılar kimi obyektlərin özlərini yox onlara müraciəti yadda saxlayırlar.

Lüğətlərlə işləmək üçün bir sıra daxili funksiyalar var.

Lüğətlərlə işləmək üçün onu yaratmaq lazımdır. Lüğəti bir neçə üsulla yaratmaq olar.

Boş lüğət yaratmaq üçünəmr sətrində içərisi boş qoşa fiqurlu mötərizələr ({}) yazmaq lazımdır.

```
>>> lugat={}
>>> lugat
{}
```

Daha sonra lüğəti yaratmaq üçün birinci üsul olaraq literaldan istifadə edək:

```
>>>lugat = {}
>>>lugat
{}
>>>lugat = {'sözlük': 1, 'lüğət': 2}
>>>lugat
{'sözlük': 1, 'lüğət': 2}
```

İkinci üsul **dict** funksiyası ilə lüğəti yaratmaqdır:

```
>>> lugat = dict(qısa='Mb', uzun='Meqabayt')
>>> lugat
{'qısa': 'Mb', 'uzun': 'Meqabayt'}
>>> lugat = dict([(1, 1), (3, 5)])
>>> lugat
{1: 1, 3: 5}
```

Üçüncü üsul fromkeys metodundan istifadə etməkdir:

```
>>>lugat = dict.fromkeys(['a', 'b'])
>>>lugat
{'a': None, 'b': None}
>>>lugat = dict.fromkeys(['a', 'b'], 23)
>>>lugat
{'a': 23, 'b': 23}
```

Dördüncü üsul lüğətlərin generatorları (bir növ siyahıların generatoruna bənzəyir) vasitəsilə.

```
>>>lugat = {a: a ** 2 for a in range(7)}
>>>lugat
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Python dilindəki lüğət haqqında təsəvvür yaratmaq üçün nümunə kimi İngilis-Azərbaycan dili lüğətini yaradaq.Lüğətdə açar kimi sətirdən istifadə ediləcək. Bildiyimiz kimi ingilis dilində olan hər bir sözə qarşı Azərbaycan dilində onun tərcüməsi var: one – bir, two – iki, three – üç, four – dörd, five – beş və s. Əgər biz İngilis-Azərbaycan lüğətini Python dilində təsvir ediriksə, bu halda ingilis sözləri açar, Azərbaycan sözləri isə qiymət olacaq, əks halda Azərbaycan sözləri açar, ingilis sözləri isə qiymət olacaq:

{'one':'bir', ' two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'}

Fiqurlu mötəzilərə diqqət yetirin, məhz onların köməyilə lüğət yaradılır.

Python dilində lüğətin sintaksisini aşağıdakı sxemlə vermək olar:

{açar:qiymət, açar:qiymət, açar:qiymət, açar:qiymət, ...}

İndi yuxarıda yaratdığımız lüğətin proqram kodunun işləməsini yoxlayaq:

```
>>> {'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'}
{'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'}
```

Diqqət eləsəniz görərsiniz ki, daxil edilən siyahı ilə yaddaşda saxlanılan və ekranda əks etdirilən siyahının ardıcıllığı eyni deyil. Öndə dediyimiz kimi lüğətdə cütlərin hansı ardıcıllıqla yerləşməsinin bir fərqi yoxdur. Əsas odur ki, axtarış açarlara görə aparılır:

```
>>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'}
>>> lugat
{'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'}
>>> lugat['four']
'dörd'
>>> lugat['one']
'bir'
```

Lüğətlər, siyahılar kimi, dəyişən tipli verilənlərdəndir. Lüğətin elementini dəyişmək, əlavə etmək və silmək (ləğv etmək) olar. Bunun (**açar:qiymət**) cütlüyündən istifadə edilir. Öndə dediyimiz kimi, əvvəlcə boş lüğət yaradılır (məsələn, **lugat = {}**). Əlavə etmə və dəyişmə eyni bir sintaksis əsasında işləyir: **lüğət[açar] = qiymət.** Əgər verilən açar artıq mövcuddursa, dəyişmə, əks halda yeni element yaratma (əlavə etmə) prosesi baş verir. Lüğətdən elementi silmək üçün **del()** funksiyasından istifadə edilir.

```
>>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'}
>>> lugat
{'five': 'beş', 'two': 'iki', 'three': 'üç', 'one': 'bir', 'four': 'dörd'}
>>> lugat['80']='həştad'
>>> lugat
{'80': 'həştad', 'two': 'iki', 'three': 'üç', 'four': 'dörd', 'five': 'beş', 'one': 'bir'}
>>> lugat['80']='səksən'
>>> lugat
{'80': 'səksən', 'two': 'iki', 'three': 'üç', 'four': 'dörd', 'five': 'beş', 'one': 'bir'}
>>> del lugat['80']
>>> lugat
{'two': 'iki', 'three': 'üç', 'four': 'dörd', 'five': 'beş', 'one': 'bir'}
```

Bu misalda əvvəlcə 5 elementi olan lüğət yaradılır. Daha sonra **lugat['80']='həştad'** kodu ilə lüğətə yeni (**'80': 'səksən'**) cütü əlavə edilir. Üçüncü addımda (**lugat['80']='səksən'**) kodu ilə açarı **'80'** olan elementin qiyməti dəyişilir. Və sonda **del lugat['80']** kodu ilə açarı **'80'** olan element silinir.

Başqa bir misala baxaq. Tutaq ki, Siz satış bazası üçün anbarda olan malların miqdarını göstərən məlumatı yadda saxlamaq istəyirsiniz. Bunun üçün lüğətdən istifadə etmək olar. Açar kimi malın adı, qiymət kimi isə malın miqdarı göstəriləcək. Əvvəlcə anbarda hansısa maldan varsa o mal haqqında məlumatı (malın adını və miqdarını)lüğətə daxil edirik. Tutaq N şirkət hansısa anda müəyyən adlı malın hamısını alır. Bu halda siyahıdan bu mal haqqında məlumatı silməyə ehtiyac yoxdur. Sadəcə olaraq həmin malın miqdarini sıfıra bərabər etmək kifayətdir. Gələcəkdə həmin malın yeni partiyası anbara daxil olanda həmin malın miqdarını dəyişmək kifayətdir.

```
>>> mallar={'dəftər':1000, 'qələm':2000, 'pozan': '500', 'xətkeş':100}
{'dəftər': 1000, 'xətkeş': 100, 'pozan': '500', 'qələm': 2000}
>>> mallar['xətkeş']=0
>>> mallar
{'dəftər': 1000, 'xətkeş': 0, 'pozan': '500', 'qələm': 2000}
```

Bu misalda xətkeş adlı malların hamısı satıldığına görə lüğətdən həmin açarlı elementin qiyməti dəyişdirilib.

Lüğətlərin çox sayda faydalı metodları var. Aşağıdakı cədvəldə bunlar haqqında məlumat verilib.

Lüğətlərin funksiyaları və metodları (üsulları)

Funksiya və metodlar	Təyinatı
lugat.clear()	Lüğəti silir. >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'} >>> lugat.clear() >>> lugat {}
lugat.copy()	Lüğətin surətini alır. >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lug=lugat.copy() >>> lug {'five': 'beş', 'three': 'üç', 'four': 'dörd', 'one': 'bir', 'two': 'iki'} >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lug=lugat >>> lugat['two']="2" >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': '2'} >>> lug {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': '2'}
lugat.fromkeys(seq[, value])	**seq** parametrindəki açarlara əsasən **value** parametrindəki qiyməti olan lüğət yaradır (susma halında **None**). >>> lugat.fromkeys("12345","test") {'4': 'test', '3': 'test', '1': 'test', '5': 'test', '2': 'test'} >>> lugat.fromkeys("12345") {'4': None, '3': None, '1': None, '5': None, '2': None}

Funksiya və metodlar	Təyinatı
lugat.get(key[, default])	Açarın qiymətini qaytarır. Əgər həmin açara uyğun element yoxdursa, default parametrindəki qiyməti qaytarır (susma halında **None**). >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lugat.get("four") 'dörd' >>> lugat.get("four","yoxdur") 'dörd' >>> lugat.get("six","yoxdur") 'yoxdur' >>> lugat.get("six") >>> >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'}
lugat.items()	(Açar, qiymət) cütlərini qaytarır. >>> lugat.items() dict_items([('five', 'beş'), ('four', 'dörd'), ('three', 'üç'), ('one', 'bir'), ('two', 'iki')])
lugat.keys()	Lüğətdəki açarları qaytarır. >>> lugat.keys() dict_keys(['five', 'four', 'three', 'one', 'two'])
lugat.pop(key[, default])	Açarı silir və qiymətini qaytarır. Əgər açar yoxdursa, **default**da olan qiyməti qaytarır. (susma halında səhv verir). >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'} >>> lugat.pop("four", "yoxlama") 'dörd' >>> lugat {'five': 'beş', 'three': 'üç', 'one': 'bir', 'two': 'iki'} >>> lugat.pop("four", "yoxlama") 'yoxlama'
lugat.popitem()	(Açar, qiymət) cütünü silir və onun qiymətini qaytarır. Əgər lüğət boşdursa, KeyError səhvini göstərir. Yadda saxlayın ki, lüğətlər nizamlanmayıb. >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lugat.popitem() ('five', 'beş')

Funksiya və metodlar	Təyinatı
	>>> lugat.popitem() ('four', 'dörd') >>> lugat.popitem() ('three', 'üç') >>> lugat {'one': 'bir', 'two': 'iki'} >>> lugat.clear() >>> lugat {} >>> lugat.popitem() Traceback (most recent call last): File "<pyshell#38>", line 1, in <module> lugat.popitem() KeyError: 'popitem(): dictionary is empty'
lugat.setdefault(key[, default])	Açarın qiymətini qaytarır. Əgər lüğətdə həmin açara uyğun element yoxdursa, onda qiyməti **default** olan açar yaradır (susma halında **None**). >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lugat.setdefault("four", "Yoxlama") 'dörd' >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'} >>> lugat.setdefault("six", "Yoxlama") 'Yoxlama' >>> lugat {'four': 'dörd', 'one': 'bir', 'five': 'beş', 'three': 'üç', 'six': 'Yoxlama', 'two': 'iki'}
lugat.update([other])	**other** lüğətinin cütlərini əlavə etməklə lüğəti yeniləyir. Mövcud açarların qiymətlərini yeniləyir. >>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lug={'five':'5', '6':'altı', '7':'yeddi', '8': 'səkkiz', '9':'doqquz'} >>> lugat {'five': 'beş', 'four': 'dörd', 'three': 'üç', 'one': 'bir', 'two': 'iki'} >>> lugat.update(lug) >>> lugat {'7': 'yeddi', 'one': 'bir', '8': 'səkkiz', 'five': '5', 'three': 'üç', '9': 'doqquz', 'four': 'dörd', '6': 'altı', 'two': 'iki'}
lugat.values()	Lüğətdəki qiymətləri qaytarır.

Funksiya və metodlar	Təyinatı
	lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'} >>> lugat.values() dict_values(['beş', 'dörd', 'üç', 'bir', 'iki'])

len funksiyası lüğətlər üçün də işləyir. Bu funksiya lüğətdəki (açar, qiymət) cütlərinin sayını qaytarır:

```
>>> lugat={'one':'bir', 'two':'iki', 'three': 'üç', 'four':'dörd', 'five':'beş'}
>>> len(lugat)
5
```

Riyaziyyatda istifadə edilən matrisləri siyahılar şəklində təsvir etmək olar. Siyahılar daha çox elementi sıfırdan fərqli olan matrisləri təsvir etmək üçün yaxşidir. Elementlərinin az hissəsi sıfırdan fərqli olan seyrək matrislər üçün lüğətdən istifadə etmək daha məqsədəuyğundur. Tutaq ki, aşağıdakı matris verilib.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Bu matrisi siyahı şəklində təsvir etsək daha çox sıfır yazmaq lazımdır.

```
matris = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

İndi isə bu matrisi lüğət kimi təsvir edək. Açar əvəzi sətir və sütunların nömrəsindən ibarət kortejdən istifadə edə bilərik. Onda bu matris belə təsvir edilər:

```
matris = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

Bizə sıfırdan fərq elementlər üçün yalnız üç cüt (açar, qiymət) lazımdır. Burada açarlar kortejdir, qiymətlər isə tam ədədlərdir.

Matrisə müraciət etmək üçün **[]** operatorundan istifadə etmək olar:

```
matris[(0, 3)]
1
```

Diqqət etsəniz görərsiniz ki, matrisin lüğət kimi təsvirinin sintaksisi bir-birinə daxil olan siyahıların təsviri sintaksisindən fərqlənir. İki tam ədədli indeks əvəzinə matrisin elementinə

müraciət etmək üçün bir açardan – iki tam ədəddən ibarət kortejdən istifadə edilir. Matrisin lüğət kimi təsviri yaddaşdan da səmərəli istifadə etməyə kömək olur.

Amma ortaya başqa bir problem çıxır. Sıfır qiymətli elementə miraciət edən zaman proqram səhv verəcək. Çünki lüğətdə matrisin sıfır qiymətli elementləri üçün cütlük yoxdur. Məsələn:

```
>>> matrix[(1, 3)]
KeyError: (1, 3)
```

Bu problemi **get** metodundan istifadə etməklə həll etmək olar:

```
>>> matrix.get((0, 3), 0)
1
```

Birinci arqument kimi açar, ikinci arqument kimi isə sıfır qiyməti götürülür. Lüğətdə açar olmayan halda **get** metodu ikinci arqumenti, yəni sıfırı qaytarır:

```
>>> matrix.get((1, 3), 0)
0
```

get metodu seyrək matrislərin elementlərinə müraciətləri yaxşılaşdırır.

Python dilində **long** tipi ixtiyari uzunluqlu tam ədədlərlə işləmək üçündür. Məhdudiyyət yalnız Sizin kompüterin yaddaşı olacaq.

long tipli qiymət yaratmaq üçün 3 üsul var.

1. **Birincisi** – nəticəsi çox böyük olub **int** tipinə yerləşməyən riyazi ifadəni hesablamalı.
2. **İkinci üsul** – ədədi əvvəlinə **L** hərfi əlavə etməklə yazmalı:

```
>>> type(1L)
```

3. Üçüncü üsul – ədədi **long** tipinə çevirmək üçün qiyməti olan **long()** çağırmalı. **long** tipini çağırmaq **int**, **float** və hətta rəqəmlər sətrini uzun ədədə çevirir:

```
>>> long(7)
7L
>>> long(3.9)
3L
>>> long('59')
59L
```

İndi isə lüğətə yazı əlavə edib və ondan yazını çıxarmağı yoxlayaq:

```
>>> lug = {1: 2, 2: 4, 3: 9}
>>> lug [1]
2
>>> lug [4] = 4 ** 2
>>> lug
{1: 2, 2: 4, 3: 9, 4: 16}
>>> lug ['1']
```

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    lug ['1']
KeyError: '1'
```

Misaldan göründüyü kimi, yeni açarla yazı daxil etdikdə lüğət genişlənir, mövcud açarla daxil edilən yazı həmin açarın qiymətini dəyişir, mövcud olmayan açarla lüğətə müraciət olduqda isə səhv verir.

Digər obyektlərlə edilən əməliyyatları (məsələn, **for** və **while** dövrləri) lüğətlər üçün də tətbiq etmək olar.

İndeksin lüğətdə olub-olmamasını yoxlamaq üçün **in** metodundan istifadə etmək olar. Aşağıda lüğətlə işləməyə bir nümunə verilib:

```
>>> tel = {'Direktor': 5501, 'Kadrlar şöbəsi': 5505}
>>> tel['Katibə'] = 5502
>>> tel
{'Kadrlar şöbəsi': 5505, 'Direktor': 5501, 'Katibə': 5502}
>>> tel['Direktor']
5501
>>> del tel['Kadrlar şöbəsi']
>>> tel['İnsan resursları']=5504
>>> tel
{'İnsan resursları': 5504, 'Direktor': 5501, 'Katibə': 5502}
>>> tel.keys()
dict_keys(['İnsan resursları', 'Direktor', 'Katibə'])
>>> 'Direktor' in tel
True
>>> 'Menecer' in tel
False
>>> tel.get("Direktor","yoxdur")
5501
>>> tel.get("Menecer","yoxdur")
'yoxdur'
```

Çoxluq (set и frozenset)

Python dilində çoxluq deyəndə ixtiyari ardıcıllıqla yerləşmiş təkrar olunmayan elementlərdən ibarət toplu - “konteyner” başa düşülür.

Gəlin bir çoxluq yaradaq:

```
>>> coxluq = set()
>>> coxluq
set()
>>> coxluq = set('salam')
>>> coxluq
{'a', 'm', 's', 'l'}
>>> coxluq = {'a', 'b', 'c', 'd'}
>>> coxluq
{'a', 'd', 'c', 'b'}
>>> type(coxluq)
<class 'set'>
>>> coxluq= {i ** 2 for i in range(10)} # çoxluqların generatoru
>>> coxluq
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
>>> coxluq = {}  # Bu səhvdir!
>>> coxluq
{}
>>> type(coxluq)
<class 'dict'>
```

Nümunədən göründüyü kimi çoxluq lüğət kimi eyni **literaldan** istifadə edir. Amma bu literalla boş çoxluq yaratmaq olmaz.

Təkrarlanan elementlərin silinməsi üçün çoxluqlardan istifadə etmək çox əlverişlidir.

```
>>> söz = ['salam', 'baba', 'salam', 'nənə']
>>> set(söz)
{'baba', 'nənə', 'salam'}
```

Çoxluqlarla bir çox əməliyyatlar (birləşmə, kəsişmə və s.) yerinə yetirmək olar.

Çoxluqların funksiyaları və metodları (üsulları)

Funksiya və metodlar	Təyinatı
len(coxluq)	Çoxluqdakı elementlərin sayı (çoxluğun ölçüsü). >>> coxluq = {'a', 'a', 'a', 'b', 'c', 'd'} >>> coxluq {'a', 'c', 'd', 'b'} >>> len(coxluq) 4
xincoxluq	**x** elementi **coxluq** çoxluğuna aiddirsə **True** (Həqiqi), əks halda **False** (Yalan) qiymət qaytarır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq {'a', 'c', 'd', 'b'} >>> 'c' in coxluq True >>> 'e' in coxluq False
coxluq.isdisjoint(diger)	**coxluq** və **diger** çoxluqları ümumi elementə malik deyilsə, True (Həqiqi), əks halda **False** (Yalan) qiymət qaytarır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'e', 'f', 'g', 'h'} >>> coxluq.isdisjoint(diger) True >>> diger = {'e', 'b', 'f', 'g'} >>> coxluq.isdisjoint(diger) False
coxluq == diger	**coxluq** çoxluğunun bütün elementləri **diger** çoxluğuna və əksinə aiddirsə, **True** (Həqiqi) qiymət qaytarır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'a', 'b', 'c', 'd'} >>> coxluq == diger True >>> diger = {'a', 'b', 'c', 'd', 'e'} >>> coxluq == diger False
coxluq.issubset(diger) və ya **coxluq<= diger**	**coxluq** çoxluğunun bütün elementləri **diger** çoxluğuna aiddirsə, **True** (Həqiqi) qiymət qaytarır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.issubset(diger) True >>> coxluq <= diger True >>> diger = {'b', 'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq <= diger False

Funksiya və metodlar	Təyinatı
coxluq.issuperset(diger) və ya **coxluq>= diger**	Əvvəlki əməliyyata oxşardır. Sadəcə olaraq əks şərt yoxlanır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.issuperset(diger) False >>> diger.issuperset(coxluq) True >>> coxluq >= diger False >>> diger >= coxluq True
coxluq.union(diger, ...) və ya **coxluq | diger | ...**	Bir neçə çoxluğun birləşməsini qaytarır. Birləşmə çoxluqların elementlərini dəyişmir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.union(diger) {'b', 'f', 'h', 'd', 'e', 'g', 'a', 'c'} >>> coxluq | diger {'b', 'f', 'h', 'd', 'e', 'g', 'a', 'c'}
coxluq.intersection(diger, ...) **və ya** **coxluq&diger& ...**	Bir neçə çoxluğun kəsişməsini qaytarır. Kəsişmə çoxluqların elementlərini dəyişmir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.intersection(diger) {'c', 'd'} >>> coxluq & diger {'c', 'd'}
coxluq.difference(diger, ...) və ya **coxluq - diger - ...**	**coxluq** çoxluğunun digər çoxluqların heç birinə aid olmayan elementləri çoxluğu. >>> coxluq = {'a', 'b', 'c', 'd', '1', '2', '3', '4'} >>> diger = {'a', 'b'} >>> diger1 = {'c', 'd'} >>> coxluq.difference(diger, diger1) {'2', '3', '1', '4'} >>> coxluq - diger - diger1 {'4', '1', '2', '3'}
coxluq.symmetric_difference (diger) və ya **coxluq ^ diger**	Çoxluqların yalnız birində rast gələn elementlərdən ibarət çoxluq. >>> coxluq = {'a', 'b', 'c', 'd', '1', '2', '3', '4'} >>> diger = {'e', 'f', 'g', 'h', '1', '2', '3', '4'} >>> coxluq.symmetric_difference(diger) {'b', 'f', 'h', 'd', 'e', 'g', 'c', 'a'} >>> coxluq ^ diger {'b', 'f', 'h', 'd', 'e', 'g', 'c', 'a'}
coxluq.copy()	Çoxluğunsurəti. >>>coxluq = {'a', 'b', 'c', 'd'}

Funksiya və metodlar	Təyinatı
	>>> coxluq.copy() {'b', 'a', 'c', 'd'}
Birbaşa çoxluğu dəyişən əməliyyatlar:	
coxluq.update(diger, ...) və ya **coxluq |= diger | ...**	Çoxluqların birləşməsi.**çoxluq** çoxluğu iki çoxluğun birləşməsindən ibarət elementlər toplusu olur. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.update(diger) >>> coxluq {'b', 'f', 'h', 'd', 'e', 'g', 'a', 'c'} >>> diger {'f', 'h', 'd', 'e', 'g', 'c'}
coxluq.intersection_update(diger, ...) və ya **coxluq&= diger& ...**	Çoxluqların kəsişməsi. **çoxluq** çoxluğu iki çoxluğun kəsişməsindən ibarət elementlər toplusu olur. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.intersection_update(diger) >>> coxluq {'d', 'c'} >>> diger {'g', 'd', 'c', 'e', 'h', 'f'}
coxluq.difference_update(diger, ...) və ya **coxluq -= diger | ...**	İki çoxluğun çıxması.**coxluq** çoxluğunu **diger** çoxluğu ilə müqayisə edir və birinci çoxluqda yalnız **coxluq** çoxluğunda olan elementləri saxlayır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.difference_update(diger) >>> coxluq {'b', 'a'} >>> diger {'g', 'd', 'c', 'e', 'h', 'f'}
coxluq.symmetric_difference_update(diger) və ya coxluq ^= diger	**coxluq** çoxluğunu **diger** çoxluğu ilə müqayisə edir və eyni vaxtda hər iki çoxluqda olan elementləri silib qalan elementlərin birləşməsindən alınan çoxluğu **coxluq** çoxluğunun elementləri kimi yadda saxlayır. >>> coxluq = {'a', 'b', 'c', 'd'} >>> diger = {'c', 'd', 'e', 'f', 'g', 'h'} >>> coxluq.symmetric_difference_update(diger) >>> coxluq {'g', 'e', 'a', 'h', 'b', 'f'} >>> diger {'g', 'd', 'c', 'e', 'h', 'f'}
coxluq.add(elem)	Çoxluğa element əlavə edir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq.add('e')

Funksiya və metodlar	Təyinatı
	>>> coxluq {'b', 'd', 'a', 'e', 'c'}
coxluq.remove(elem)	Çoxluqdan elementi silir. Əgər belə element yoxdursa, **KeyError** səhvi verir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq.remove('e') Traceback (most recent call last): File "<pyshell#29>", line 1, in <module> coxluq.remove('e') KeyError: 'e' >>> coxluq.remove('b') >>> coxluq {'d', 'a', 'c'}
coxluq.discard(elem)	Əgər element çoxluqda varsa onu silir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq.discard('e') >>> coxluq.discard('b') >>> coxluq {'d', 'a', 'c'}
coxluq.pop()	Çoxluqda birinci elementi silir. Çoxluğun elementləri nizamlanmadığından hansı elementin birinci olmasını dəqiq demək olmaz. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq {'b', 'd', 'a', 'c'} >>> coxluq.pop() 'b' >>> coxluq {'d', 'a', 'c'}
coxluq.clear()	Çoxluğutəmizləyir. >>> coxluq = {'a', 'b', 'c', 'd'} >>> coxluq {'b', 'd', 'a', 'c'} >>> coxluq.clear() >>> coxluq set()

frozenset

set və **frozenset** prinsipcə bir-birinə oxşayır. **set** ilə **frozenset** arasında fərq yalnız ondadır ki, **set** dəyişən tipdir, **frozenset** isə yox. Onları siyahı və kortejlə müqayisə etmək olar.

```
>>> a = set('qwerty')
>>> b = frozenset('qwerty')
>>> a == b
True
>>> True
True
>>> type(a - b)
<class 'set'>
>>> type(a | b)
<class 'set'>
>>> a.add(5)
{5, 'w', 'e', 'y', 'q', 't', 'r'}
```

if-elif-else ifadəsi (əmri), həqiqiliyin yoxlanması, üçyerli if/else ifadəsi

if-elif-else şərt əmri (bu əmri bəzən budaqlanma operatoru da adlandırırlar) – Python dilinin əsas əsas seçim alətidir. Sadəcə desək, o, şərt yoxlanan anda dəyişənlərin qiymətindən asılı olaraq hansı hərəkətlərin yerinə yetiriləcəyini seçir.

if əmrinin sintaksisi

Əvvəlcə şərti ifadəsi olan **if** hissəsi yazılır, daha sonra bir və ya daha çox vacib olmayan **elif** hissəsi gəlir və nəhayət sonda yenə də vacib olmayan **else** hissəsi gəlir. **if** şərt operatorunun ümumi forması aşağıdakı kimi olur:

```
if şərt1:
    hal1
elif şərt2:
    hal2
else:
    hal3
```

Yazılışda bir detala fikir verin ki, **elif** və **else:** yazıları sürüşməsiz yazılıbdır.
Sadə misal (1 – həqiqi olduğu üçün 'true' yazmalı):

```
>>> if 1:
        print ('true')
else:
        print ('false')
true
```

Nisbətən çətin misal:

```
>>> a = 7
>>> if a < -5:
    print('Aşağı')
elif -5 <= a <= 5:
    print('Orta')
else:
    print('Yüksək')

Yüksək
```

Misalı bir qədər də çətinləşdirək. **a** dəyişəninin qiyməti öncədən məlum deyil, ona qiymət istifadəçi tərəfindən **(a = int(input()))** veriləcək. Bu proqram fraqmentini Python redaktorunda yazaq. Bunun **Ctrl+N** klavişlərini sıxmaqla Python redaktorunu çağırıb aşağıdakı proqram mətnini orada yazmalıyıq:

```
a = int(input())
if a<-5:
    print('Aşağı')
elif -5<=a<=5:
    print('Orta')
else:
    print('Yüksək')
```

Daha sonra **F5** klavişini sıxmaqla proqramı yerinə yetirək. Proqram Python örtüyündə aşağıdakı görüntü verəcək. Nümunədə **a** dəyişəni üçün 7 qiyməti daxil edilib.

```
>>>================= RESTART =================
>>>
7
Yüksək
```

if-elif-else ifadəsi bir konstruksiya kimi bir çox dillərdə olan **switch-case** konstruksiyasına əlavədir.

Üçyerli if/else ifadəsi

Aşağıda bir proqram fraqmenti verilib.

```
if X:
    A = Y
else:
    A = Z
```

Həcmcə kiçik olsa da, bu proqram 4 sətir tutur. Belə hallar üçün **if/else** ifadəsi nəzərdə tutulub:

```
A = Y if X else Z
```

Bu proqram parçasında əgər X həqidirsə, interpretator Y ifadəsini, əks halda Z ifadəsini yerinə yetirir.

```
>>> A = 't' if 'spam' else 'f'
>>> A
't'
```

for və while dövrləri, break və continue operatorları, else sözü

İndi isə dövr operatorları ilə tanış olaq.

while dövrü

while–Python dilində ən universal dövrlərdən biridir. Onun işləmə sürəti zəifdir. Dövrün gövdəsi dövrün şərti həqiqi olana qədər yerinə yetiriləcək.

```
>>> i = 4
>>> while i < 16:
        print(i)
        i = i + 2

4
6
8
10
12
14
```

for dövrü

for dövrü nisbətən çətindir, daha az universaldır, amma **while** dövrünə nəzərən daha çox sürətli işləyir. Bu dövr ixtiyari iterasiya olunan obyekt üçün (məsələn, sətir və ya siyahı) tətbiq olunur. Yazılış qaydası belədir:

for *dəyişən* in [obyekt] [range(start, stop, step)]

```
>>> for i in ' Salam dost ':
print(i * 3, end='')

SSSaaalllaaammm   dddooosssttt
```

Yuxarıdakı misalda sətirdəki mətnin hər bir hərfi 3 dəfə çap olunur.

continue operatoru

continue operatoru dövrün (**for** və ya **while**) qalan hissəsini buraxaraq dövrün növbəti mərhələsini başlayır.

```
>>> for i in 'Salam dost':
        if i == 'l':
                continue
        print(i * 2, end='')

SSaaaamm  ddoosstt
```

Yuxarıdakı misalda sətirdə "**l**" hərfi rast gələn kimi dövrün qalan əmrləri yerinə yetirilmir, yəni "**l**" hərfi çap olunmur və növbəti mərhələsi (addımı) başlayır.

break operatoru

break operatoru dövrü yarıda kəsir.

```
>>> for i in 'Salam dost':
        if i == 'o':
                break
        print(i * 2, end='')

SSaallaamm  dd
```

Yuxarıdakı misalda sətirdə **"o"** hərfi rast gələn kimi dövr kəsilir.

else sözü

for və ya **while** dövrlərində **else** sözü dövrdən "təbii" ardıcıllıqla, yoxsa **break** əmri ilə çıxmağı yoxlayır. **else** daxilindəki əmrlər bloku dövrdən çıxış break əmrinin köməyilə olmadıqda işləyir.

```
>>> for i in 'Salam dost':
        if i == 'ü':
                break
        else:
                print('Sətirdə ü hərfi yoxdur')
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
```

```
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
>>> for i in 'Salam dost':
        if i == 'l':
                break
        else:
                print('Sətirdə ü hərfi yoxdur')

Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
Sətirdə ü hərfi yoxdur
```

Yuxarıdakı misaldan göründüyü kimi **break** operatoru işləyənə qədər dövr nə qədər addım işləmişsə, **else** daxilindəki əmrlər bloku da o qədər dəfə işləyəcək.

Açar sözlər

Söz	Təyinatı
False	Yalan.
True	Həqiqi.
None	"boş" obyekt.
and	məntiqi HƏ.
with / as	kontekst meneceri.
assert	əgər şərt yalandırsa, şərt istisna doğurur.
break	dövrdən çıxış.
class	metod (üsul) və atributlardan ibarət istifadəçi tipi.
continue	dövrün növbəti iterasiyasına keçid.
def	funksiyanın təyini.
del	obyektin silinməsi (ləğv edilməsi).
elif	əks halda əgər.
else	əks halda.
except	istisna tutmaq.
finally	**try** əmri ilə birlikdə istisnanın olub olmamasından asılı olmayaraq əməliyyatı yerinə yetirir.
for	for dövrü.
from	moduldan bir neçə funksiyanın importu.
global	funksiya daxilində qiymət mənimsədilmiş dəyişəninin qiyməti funksiyadan kənar da əlçatan edir.
if	əgər.
import	modulun importu.
in	– daxilolmanı yoxlayır
is	2 obyekt yaddaşda eyni bir yerə müraciət edirmi.
lambda	anonim funksiyanın təyin edilməsi.
nonlocal	funksiya daxilində qiymət mənimsədilmiş dəyişəninin qiyməti əhatə olunduğu proqramda əlçatan edir.
not	məntiqi YOX.
or	məntiqi VƏYA.
pass	heç nə etməyən konstruksiya.
raise	istisna yaratmaq.
return	nəticəni qaytarmaq.
try	istisnaları ələ keçirməklə əmrləri yerinə yetirmək.
while	while dövrü.
yield	funksiya generatorun təyin edilməsi.

Python dilində istisnalar. İstisnaları emal etmək üçün try - except konstruksiyası

Müasir proqramlarda verilənlərin ötürülməsi həmişə deyilən kimi hamar olmur. Xüsusi halların emalı üçün (sıfıra bölmə, mövcud olmayan fayldan oxumaq və s.)verilənlərin bir tipi olan istisnalardan(exceptions) istifadə olunur. İstisnalar proqramçını səhvlər haqqında məlumatlandırmaq üçün lazımdır.

Buna bariz ən sadə sıfıra bölmədir:

```
>>> 23/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
  23/0
ZeroDivisionError: division by zero
```

Bu halda interpretator səhv kimi bildirir ki, sıfıra bölmə halı var.

Başqa bir istisnaya baxaq:

```
>>> 2 + '1'
Traceback (most recent call last):
 File "<pyshell#0>", line 1, in <module>
  2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> int('adi mətn')
Traceback (most recent call last):
 File "<pyshell#1>", line 1, in <module>
  int('adi mətn')
ValueError: invalid literal for int() with base 10: 'adi mətn'
```

Bu 2 misalda uyğun olaraq **TypeError** və **ValueError** isitsnaları generasiya olur. İsitsnalardakı informasiya bizə vəziyyət haqqında tam informasiya verir.

Daxili istisnalar

Python dilində bütün istisnalar baza istisnaları əsasında alınır.
Python dilində rast gələn daxili istisnaların təbiəti ilə tanış olaq. Proqramçılar öz proqramlarında fərqli istisnalar nəzərdə tuta bilər. Aşağıda baza istisnaları və onların qısa şərhi iearxiyaya əsaslanaraq verilib:

- ✓ **BaseException** – *Digər istisnaların başlanğıc kimi istinad etdiyi baza isitsnalar.*
  - ► **SystemExit** –*Proqramdan çıxan zaman* ***sys.exit*** *funksiyası tərəfindən yaranan istisna.*
  - ► **KeyboardInterrupt** –*Proqram istifadəçi tərəfindən müdaxilə edilən zaman (adətən****Ctrl+C****klavişlərinin birgə sıxılması)yaranan istisna.*
  - ► **GeneratorExit** –***generator*** *obyektinin* ***close*** *metodunu çağıran zaman yaranır.*
  - ► **Exception** –*Burada isə sistem istisnaları (yaxşısı budur ki, onlara toxunmayasınız) tamamilə bitir və onlarla işləmək imkanı olan adi istisnalar başlayır.*
  - ► **StopIteration** –*Əgər iteratorda daha element yoxdursa,* ***next*** *daxili funksiyası ilə yaranır.*

- **ArithmeticError** –*Riyazi səhv.*
  - **FloatingPointError** –*Sürüşkən vergüllə əməliyyat uğurlu aparılmayanda yaranır. Praktikada çox az halda rast gəlinir.*
  - **OverflowError** –*Riyazi əməliyyatın nəticəsi təqdim olunmaq üçün həddindən böyükdürsə, yaranır. Adətən adi tam ədədlərlə işləyən zaman əmələ gəlmir (çünki Python böyük ədədlərlə işləyə bilir). Amma digər hallarda əmələ gələ bilir.*
  - **ZeroDivisionError** –*Sıfıra bölmə.*
- **AssertionError** –***assert****funksiyasında ifadə səhvdir.*
- **AttributeError** –*Obyekt bu atributa (qiymət və ya metoda) sahib deyil.*
- **BufferError** –*buferlə bağlı əməliyyat yerinə yetirilə bilməz.*
- **EOFError** –*Funksiya faylın sonuna çatıb və istədiyini oxuya bilməyib.*
- **ImportError** –*Modul və ya onun atributu import oluna bilmədi.*
- **LookupError** –*Korrekt olmayan indeks və ya açar.*
  - **IndexError** –*indeks elementlər diapazonuna daxil deyil.*
  - **KeyError** –*Mövcud olmayan açar (lüğətdə, çoxluqda və ya digər obyektdə.*
- **MemoryError** –*Kifayət qədər yaddaş çatmır.*
- **NameError** –*Bu adda dəyişən tapılmadı.*
  - **UnboundLocalError** –*Funksiyada əvvəl təyin olunmamış lokal dəyişənə müraciət olub.*
- **OSError** –*Sistemlə bağlı səhv.*
  - **BlockingIOError** -
  - **ChildProcessError** –*alt proseslə əməliyyat zamanı uğursuzluq.*
  - **ConnectionError** –*qoşulmalar ilə əlaqədar istisnalar üçün baza sinfi.*
  - **FileExistsError** –*mövcud olan faylı və ya qovluğu yaratmaq cəhdi.*
  - **FileNotFoundError** –*fayl və ya qovluq mövcud deyil.*
  - **InterruptedError** –*Sistem çağırışı daxil olan siqnalla kəsilib.*
  - **IsADirectoryError** –*fayl gözlənilirdi, qovluq oldu.*
  - **NotADirectoryError** - *qovluq gözlənilirdi, fayl oldu.*
  - **PermissionError** –*Müraciət hüququ çatmır.*
  - **ProcessLookupError** –*Göstərilən proses mövcud deyil.*
  - **TimeoutError** –*Gözləmə vaxtı bitdi.*
- **ReferenceError** –*Zəif müraciətli atributa müraciət cəhdi.*
- **RuntimeError** –*İstisna digər kateqoriyalardan heç birinə uyğun gəlməyəndə meydana çıxır.*
  - **NotImplementedError** –*Sinfin abstrakt metodları alt siniflərdə yenidən təyinetmə tələb edi*
- **SyntaxError** –*Sintaksissəhv.*
  - **IndentationError** –*Düzgün olmayan sürüşmələr.*
    - **TabError** – *Tabulyasiya və probellərin sürüşmələrində səhv.*
- **SystemError** - *Daxili səhv.*
- **TypeError** –*Funksiya tipəuyğun olmayan obyektə tətbiq edilib.*

- **ValueError** –*Funksiya tipi düzgün olan arqument alır, amma qiymәti korrekt olmur.*
    - **UnicodeError** –*Unicodun sәtirdә kodlaşdırılma/dekodlaşdırılma ilә bağlı sәhv.*
        - **UnicodeEncodeError** - *Unicod kodlaşdırılması ilә bağlıistisna.*
        - **UnicodeDecodeError** - *Unicodun dekodlaşdırılması ilә bağlıistisna.*
        - **UnicodeTranslateError** - *Unicodun çevrilmәsi ilә bağlıistisna.*
- **Warning** –*Xәbәrdarlıq kateqoriyası üzrә baza sinif.*
    - **DeprecationWarning**–*Köhnә funksiyalar haqqında baza istisnaları.*
    - **PendingDeprecationWarning**–*Gәlәcәkdәverilәnlәr bazası köhnәlmiş hesab edilәn istisnalar sinfi üçün xәbәrdarlıqlar.*
    - **RuntimeWarning**–İcranın şübhәli aparması haqqında xәbәrdarlıqlar üçün baza sinfi.
    - **SyntaxWarning**–Şübhәli sintaksisli haqqında xәbәrdarlıqlar üçün baza sinfi.
    - **UserWarning**–İstifadәçi kodu üzrә generasiya olan xәbәrdarlıqlar üçün baza sinfi.
    - **FutureWarning**–Gәlәcәkdә semantik dәyişilәn konstruksiyalar haqqında xәbәrdarlıq üçün baza sinfi.
    - **ImportWarning**–Modulun importu zamanı mümkün sәhvlәr üçün baza sinfi.
    - **UnicodeWarning**–Unicod ilә bağlı xәbәrdarlıqlar üçün baza sinfi.
    - **BytesWarning**–Bayt vә ByteArray ilә bağlı xәbәrdarlıqlar üçün baza sinfi.
    - **ResourceWarning**–İstifadә edilәn resurslarla bağlı xәbәrdarlıqlar üçün baza sinfi.

Artıq istisnaların nә vaxt vә hansı şәraitdә baş verdiyini bildikdәn sonra biz onları emal edә bilәrik. Dİgәr dillәrdәn fәrqli olaraq Python dilindә istisnalar alqoritmi sadәlәşdirmәk üçün xidmәt edir.

İstisnaları emal etmәk üçün **try - except** konstruksiyasından (operatorundan) istifadә edilir.

try – exceptkonstruksiyasından istifadә edәrkәn proqramçı ilk növbәdә belә düşünmәlidir: "Yoxlayaram, әgәr alınmasa except kodu yerinә yetirilәcәk".

Gəlin bir misala baxaq:

```
>>>try:
... k=1/0
... exceptZeroDivisionError:
... k=0
...
>>>print(k)
0
```

Biz **try** blokunda istisna yaradan əməliyyat (sıfıra bölmə) aparırıq, **except** blokunda isə həmin istisnanı tuturuq. Bu halda həm istisnalar, həm də ondan əmələ gələnlər tutulur. Məsələn, **ArithmeticError** istisnasını tutanda biz eyni zamanda **FloatingPointError**, **OverflowError** və **ZeroDivisionError** istisnalarını da tutmuş oluruq.

```
>>>try:
...     k = 1 / 0
... except ArithmeticError:
...     k = 0
...
>>>print(k)
0
```

except əmri arqumentsiz də işlənə bilir,yəni **except** bütün istisnaları (həm klaviaturadan, həm sistem çıxışından və s.) tutur. Bu halda **except** əmri faktiki istifadə edilmir, əvəzində **except Exception** işlədilir. Adətən proqramın işləkliyini yoxlamağı sadələşdirmək üçün çox vaxt bu üsuldan istifdadə edilir.

Daha iki əmrlə tanış olaq, **finally** və **else**. **Finally** istisnanın olub-olmamsından asılı olmayaraq əmrləri yerinə yetirir (təcili iş görmək üçün, məsələn, faylı bağlamaq). **else** əmriistisna olmayan halda işləyir.

```
>>> f = open('1.txt')
>>> ints = []
>>>try:
...     for line in f:
...         ints.append(int(line))
... except ValueError:
...     print('Bu ədəd deyil. Biz çıxırıq.')
... except Exception:
...     print('Bəs bu nədir?')
... else:
...     print('Hər şey yaxşıdır.')
... finally:
...     f.close()
...     print('Mən faylı bağladım.')
...     # Ardıcıllıq belə olmalıdır: try, группа except, sonra else, və nəhayət
finally.
...
```

```
Bu ədəd deyil. Biz çıxırıq.
Mən faylı bağladım.
```

Baytlar (bytes və bytearray)

Python dilində bayt sətri deyəndə nə başa düşülür? Bayt sətirləri adi sətirlərə çox oxşayırlar. Amma bir qədər fərqləri var. Bəs onlar nədədir?

Bayt nədir? Bayt rəqəmsal informasiyanı yadda saxlayıb emal etmək üçün kiçik ölçü vahidir. Baytlar ardıcıllığı hansısa informasiyadır (mətn,şəkil, musiqi və s...).

Gəlin bayt sətri yaradaq:

```
>>> b'bytes'
b'bytes'
>>> 'Baytlar'.encode('utf-8')
b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'
>>> bytes('bytes', encoding = 'utf-8')
b'bytes'
>>> bytes([50, 100, 76, 72, 41])
b'2dLH)'
```

Bildiyimiz kimi **bytes** funksiyası 1 bayt 0 və 255 diapazonunda olan ədədləri qəbul edir və **chr**funksiyasının tətbiqi ilə alınan baytları qaytarır.

```
>>> chr(50)
'2'
>>> chr(100)
'd'
>>> chr(76)
'L'
```

Baytlarla nə etmək olar? Bütün sətir metodları bayt sətri ilə işləsə də, onunla çox iş görmək olmur. O daha çox fayllarla işləyən zaman, kodlaşdırma və dekodlaşdırma zamanı istifadə edilir. Sətrə çevirmək üçün **decode** metodundan istifadə edilir.

Bytearray

Python dilində **Bytearray** baytlar massividir. **bites** tipindən onunla fərqlənir ki, dəyişiləndir.

```
>>> b = bytearray(b'hello world!')
>>> b
bytearray(b'hello world!')
>>> b[0]
104
>>> b[0] = b'h'
Traceback (most recent call last):
  File "", line 1, in
    b[0] = b'h'
TypeError: an integer is required
>>> b[0] = 105
>>> b
bytearray(b'iello world!')
>>>for i in range(len(b)):
...     b[i] += i
...
>>> b
bytearray(b'ifnos%}vzun,')
```

Fayllar. Fayllarla iş

Bütün proqramlaşdırma dilləri kimi Python dilinində də vacib funsiyalarından biri də fayllarla işdir. Fayllarla iş deyəndə faylın açılması/bağlanması, oxunması və yazılması nəzərdə tutulur.

Fayllarla işə başlamaq üçün ilk növbədə onu açmaq lazımdır. Bunun üçün daxili **open** funksiyası nəzərdə tutulub. Məsələn:

```
f = open('text.txt', 'r')
```

Python dilində **open** funksiyasının ümumi yazılışı belədir:

open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True)
Burada:
file – *faylın adı;*
mode – *faylı açmaq istədiyimiz rejim;*
buffering –*buferləşdirmə;*
encoding - *kodlaşdırma;*
errors –*səhvlər;*
newline –*yeni sətrə keçid;*
closefd –*faylı bağlama əlaməti.*

Yazılışdan göründüyü kimi **open** funksiyasının arqumentləri çoxdur. İş prosesində ən çox 3 parametr daha istifadə edilir. Bunlar: faylın adı (ünvan nisbi və ya mütləq ola bilər), rejim və encoding arqumentləridir. Aşağıdakı cədvəldə faylı açan zaman istifadə edilən rejimlər (**mode**) göstərilib.

Rejim	Təyinatı
‘r’	Oxumaq üçün açılır (susma halında işlədilir).
‘w’	Yazmaq üçün açılır, faylın məzmunu silinir, əgər fayl yoxdursa, yeni fayl yaradılır.
‘x’	Əgər fayl yoxdursa, yazmaq üçün açılır, əks halda istisna yaranır.
‘a’	Əlavə etmək üçün açılır, informasiya faylın sonuna əlavə edilir.
'b'	İkilik rejimdə açılır.
't'	Mətn rejimində açılır (susma halında istifadə edilir).
‘+’	Oxumağa və yazmağa açılır.

Rejimlər birləşdirilə bilər. Məsələn: **'rb'**–ikilik rejimdə oxumaq. Susma halında rejim **'rt'** qiymətini alır.

Sonuncu vacib arqument **encoding** arqumentidir və faylı mətn rejimində oxumaq üçün lazımdır. Bu arqument kodlaşdırmanı verir.

Fayldan oxumaq

Tutaq ki, biz faylı açdıq. Bu o deməkdir ki, fayl işləməyə hazırdır. Biz artıq ondan informasiya ala bilərik. Bu hal üçün bir neçə üsul var, ancaq 2 üsul daha çox əhəmiyyət kəsb edir.

Birinci üsul –**read** metodu, əgər heç bir arqument verilməyibsə, faylı bütöv oxuyur, arqument olan zaman arqumentdə göstərilən miqdarda (məsələn, **n**) sayda simvol oxunur.

```
>>>f = open('text.txt')
>>>f.read(1)
'H'
>>>f.read()
‘ello world!\nThe end.\n\n’
```

İkinci üsul – **for** əmrindən istifadə etməklə faylı sətir-sətir oxumaqdır:

```
>>>f = open('text.txt')
>>>for line in f:
...    line
...
'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

Fayla yazmaq

İndi isə fayla yazmağa baxaq. Fayla aşağıdakı siyahını yazmağa çalışaq:

```
>>> m = [str(i)+str(i-1) for i in range(10)]
>>> m
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98']
```

Faylı yazmaq üçün açaq:

```
>>> f = open('text.txt', 'w')
```

Fayla yazmaq **write** metodu ilə aparılır:

```
>>> for index in m:
        f.write(index + '\n')
4
3
3
3
3
3
3
3
3
3
```

Nəticədə **write** metodunun yazılan simvolların sayını qaytarır.

Faylla işi bitirdikdən sonra **close** metodu vasitəsilə onu mütləq bağlamaq lazımdır:

```
>>> f.close()
```

İndi isə yaradılmış fayl əsasında əvvəlki siyahını bərpa edək. Faylı oxumaq rejimində açırıq və sətri oxuyuruq.

```
>>> f = open('text.txt', 'r')
>>> m = [line.strip() for line in f]
>>> m
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98']
>>> f.close()
```

Göründüyü kimi biz həmin siyahını aldıq.

PEP 8 –Python dilində kod yazmaq üçün təlimat

Python dilində proqram kodu yazarkən sürüşmənin hər bir səviyyəsi üçün 4 probeldən (boşluqdan) istifadə edin.

Çoxsətirli konstruksiyalarda bağlayıcı yumru/kvadrat/fiqurlu mötərizə siyahının sonuncu sətrinin probel olmayan birinci simvolunun altında yerləşə bilər, məsələn:

```
reqem=[
   1,2,3,
   4,5,6,
   7,8,9,0
   ]
```

Başqa variant bağlayıcı mötərizə çoxsətirli konstruksiyalarda birinci simvolun yerində ola bilər:

```
reqem=[
        1,2,3,
        4,5,6,
        7,8,9,0
]
```

Soruşa bilərsiniz, sürüşmələri probellə, yoxsa tabulyasiya vermək məsləhətdir? Probel sürüşmələrdə daha məqsədəuyğundur.

Tabulyasiyadan istifadə yalnız tabulyasiya vasitəsilə sürüşmələrlə yazılan kodları dəstəkləmək üçün istifadə edilir.

Python 3 dilində bir əmr daxilində sürüşmələrdə eyni vaxtda həm tabulyasiya, həm də probeldən istifadə etmək qadağandır.

Sətrin uzunluğunu maksimum 79 simvolla məhdudlaşdırın.

Az struktur məhdudiyyətləri olan daha böyük mətn blokları üçün sətrin uzunluğunu maksimum 72 simvolla məhdudlaşdırın.

Proqram mətninin eninin qısaldılması eyni vaxtda bir proqramın iki versiyasını redaktorun pəncərəsində müqayisə etməyi asanlaşdırır.

Bəzi əmrlər daha uzun sətirdə veriləndə daha oxunaqlı olur. Bu qrup əmrlər üçün sətrin uzunluğunu 99 simvoladək artıqmaq normaldır. Burada da şərh və sənədləşmənin sətirlərinin uzunluğu 72 simvoldan artıq olmamalıdır.

Python dilində əmrin yazılışında mötərizələrdən istifadə ediləndə, mötərizədaxili mətni tərsinə çəpinə xətt ("\") vasitəsilə bir neçə sətrə bölmək olar. Məsələn:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
file_2.write(file_1.read())
```

Yüksək səviyyəli funksiyaları və siniflərin təyin edilməsini iki boş sətirlə ayırın.

Sinif daxilində metodların (üsulların) təyin edilməsi bir boş sətirlə ayrılır.

Funksiyalarda məntiqi bölmələri ayırmaq üçün boş sətirlərdən istifadə edin.

Python 3 dilində kodlaşdırmakimi UTF-8 istifadə edilir.

Python 3.0 versiyasından başlayaraq qəbul edilib ki, standart kitabxanalarda bütün identifikatorlar ASCII simvolları ilə adlanmalıdır.

Hər bir import ayrıca sətirdə verimməlidir.

Düzgün yazılış:

```
import os
import sys
```

Səhv yazılış:

```
~~import sys, os~~
```

İmportlar faylın əvvəlində modullara şərh və sənədləşmə sətirlərində sonra və sabitlərin təsvirindən əvvəl yerləşdirilir.

İmportlar aşağıdakı qaydada qruplaşdırılmalıdır:

1. Standart kitabxanalardan importlar,

2. Kənar kitabxanalardan import,

3. Cari layihənin modullarının importu.

Hər import qrupu arasında boş sətir qoyun.

İmportdan sonra **__all__**spesifikasiyasını yazın.

Aşağıdakı hallarda probeldən istifadə etməyin:

- Yumru, kvadrat və ya fiqurlu mötərizələrin içində.

Düzgün yazılış:

```
sahe(ham[1], {eggs: 2})
```

Səhv yazılış:

```
~~sahe( ham[ 1 ], { eggs: 2 } )~~
```

- Vergül, nöqtə-vergül və ya iki nöqtədən əvvəl:

Düzgün yazılış:

```
if x == 4: print(x, y); x, y = y, x
```

Səhv yazılış:

```
if x == 4 : print(x , y) ; x , y = y , x
```

- Funksiyanı çağıran zaman ondan sonra arqumentlərin siyahısının başladığı açıq mötərizədən əvvəl:

Düzgün yazılış:

```
gun(1)
```

Səhv yazılış:

```
gun (1)
```

- Ondan sonra indeks və ya kəsiklərin başladığı açıq mötərizədən əvvəl:

Düzgün yazılış:

```
dict['key'] = list[index]
```

Səhv yazılış:

```
dict ['key'] = list [index]
```

- Bir-biri ilə nizamlamaq üçün mənimsətmə operatorunun ətrafında birdən çox probeldən istifadə:

Düzgün yazılış:

```
x = 1
y = 2
long_variable = 3
```

Səhv yazılış:

```
x             = 1
y             = 2
long_variable = 3
```

Digər tövsiyyələr

- Aşağıdakı binar operatorları həmişə hər tərəfdən bir probellə əhatə edin: mənimsətmə (**=,+=, -=** və s.), müqayisə (**==, <, >, !=, <>, <=, >=, in, not in, is, is not**), məntiqi (**and, or, not**).

- Müxtəlif priotetli operatorlardan istifadə edilirsə, onda aşağı prioritet operatorları ətrafında probel əlavə etməyə cəhd edin. Öz qərar istifadə edin, lakin heç vaxt bir çox yer istifadə və həmişə ikili operator hər tərəfdən fəzalarında eyni sayda istifadə edin. Binar operatorların hər iki tərəfində heç vaxt birdən artıq probel qoymayın və bu probelin sayı hər iki tərəfdən eyni olmalıdır.

  Düzgün yazılış:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

  Səhv yazılış:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Proqram nümunələri

Məsələ. İki **a** və **b** ədədi verilib. Elə etmək lazımdır ki, onların qiymətləri yerlərini dəyişsin.

Məsələnin qoyuluşu: Müəyyən qiymətə malik iki dəyişən var. Tutaq ki, **a** dəyişəninin qiyməti **x**, **b** dəyişəninin qiyməti isə **y**-dir. Tələb olunur ki, **a** dəyişəninin qiyməti **y**-ə, **b** dəyişəninin qiyməti isə **x**-ə bərabər olsun.

Həlli:

Həlli: Digər proqramlaşdırma dillərində bu məsələni həll etmək üçün aralıq üçüncü **c** dəyişənindən istifadə edilirdi. Bunun üçün aşağıdakı əməliyyalar yerinə yetirilirdi:

c=a ➔ a=b ➔ b=c

Python proqramlaşdırma dilində bu məsələ çox sadə yolla öz həllini tapır. Öncə **a** və **b** dəyişənlərinə qiymət mənimsədilir. Daha sonra onların qiymətləri dəyişdirilir.

Python dilində proqramın mətni belə olacaq:

```
# İki ədədin yerinin dəyişdirilməsi
#
a=input("a dəyişəninin qiymətini daxil edin: ")
b=input("b dəyişəninin qiymətini daxil edin: ")
(a,b)=(b,a)
print("a-nın yeni qiyməti" , a , " \n " , "b-nin yeni qiyməti ",b)
```

'\n' kombinasiyası print əmrində yeni sətrə keçid verir.

Bir şeyə diqqət edək. **input** əmri daxil olunan məlumatı sətir formatında qaytarır. Daxil olunan informasiyanı ədəd formatında daxil etmək üçün onu sətir formatından ədəd formatına çevirmək lazımdır (məsələn, **int** funksiyası vasitəsilə).

Məsələ. Adı və soyadı daxil edib onları aralarında probel qoymaqla birləşdirməli və alınan sətrin uzunluğunu hesablayıb nəticələri ekranda çap etməli.

Məsələnin qoyuluşu: Şərtə uyğun olaraq **ad** və **soyad** dəyişənlərinə daxil edilir. Tələb olunur ki, **ad** və **soyad** dəyişənləri aralarında probel olmaqla birləşdirilib **c** dəyişəninə mənimsədilsin və nəticələr ekranda çap edilsin.

Python dilində proqram belə olacaq:

```
# İki ədədin cəminin tapılması
#
ad=input("Adınızı daxil edin: ")
soyad=input("Soyadınızı daxil edin: ")
c=ad+" "+soyad
d=len(c)
print ("Sizin Sizin adınız və soyadınız -",c,d," simvola malikdir.")
```

Diqqət etsəniz görərsiniz ki, ədədlərdən fərqli olaraq sətirlərin toplanması onların birləşməsi ilə nəticələnir. Öndə deyildiyi kimi, bu proses **konkatenasiya** adlanır.

Məsələ. İki **a** və **b** ədədini daxil edib onların cəmini tapın.

Məsələnin qoyuluşu: Müəyyən qiymətə malik iki **a** və **b** dəyişənin daxil edilir. Tələb olunur ki, **a** və **b** ədədlərinin cəmi **c** dəyişəninə mənimsədilsin və ekranda çap edilsin.

Python dilində proqram belə olacaq:

```
# İki ədədin cəminin tapılması
#
a=int(input("Birinci ədədi daxil edin: "))
b=int(input("İkinci ədədi daxil edin: "))
c=a+b
print ("Sizin daxil etdiyiniz birinci ədəd -",a,", ikinci ədəd -",b,", onların cəmi isə -
",c," olacaq.")
```

Məsələ. Əmək haqqı və gəlir vergisinin faiz dərəcəsi məlumdur. Gəlir vergisinin və ələ alınacaq pulun həcmini təyin etməli.

Məsələnin qoyuluşu: İlkin verilən əməkhaqqının məbləği (ədədlə ifadə edilən **maash** dəyişəni) və gəlir vergisindən faiz dərəcəsi (ədədlə ifadə edilən **faiz** dəyişəni) olacaq. Verginin həcmi (**vergi** dəyişəni) **maash*faiz/100** düsturu ilə, ələ alınacaq məbləğ isə (**nagd** dəyişəni) **maash-vergi** düsturu ilə hesablanır.

Python dilində proqram belə olacaq:

```
# Əmək haqqının hesablanması
#
maash=int(input("Maaş: "))
faiz=int(input("Vergidən % dərəcəsi: "))
vergi=(maash*faiz)/100
nagd=maash-vergi
print("Ələ alınacaq məbləğ: ",nagd)
print("Vergi: ",vergi)
```

Burada bütün ədədləri tam götürsək, nəticə dəqiq alınmaya bilər. Buna görə **vergi** dəyişəninə həqiqi ədəd (**float** funksiyası) mənimsədilir.

Məsələ. Sətirdəki böyük və kiçik hərflərin sayının faizlə həcmini təyin etməli.

Məsələnin qoyuluşu: **setir** adlı sətrə müxtəlif reqistrdə (böyük və kiçik) hərflər daxil edilmişdir. Həmin sətirdəki böyük və kiçik hərflərin sayını tapıb onların sətrin ümumi uzunluğunda neçə faiz təşkil etdiyini tapmalı.

(Oxşar məsələ http://younglinux.info/python/task/string saytında var)

Bildiyimiz kimi kodlaşdırmada simvollar (o cümlədən hərflər) nizamlanıb, yəni hərflərin kodları üçün növbəti bərabərsizlik doğrudur:**'a' < 'b'**. Biz yalnız latın hərflərindən istifadə edəcəyik. Ona görə də hərflərin **'A'-'Z'** və **'a'-'z'** diapazonlarına düşməsini yoxlamaqla məqsədimizi çatmış olarıq. Növbəti simvol **'A'-'Z'** diapazonuna aiddirsə, böyük hərf, **'a'-'z'** diapazonuna aiddirsə kiçik hərfdir. Bu diapazonların heç birinə aid olmayan simvol hərf deyil (digər bir işarədir-probel, rəqəm, durğu işarəsi və s.).

Burada şərt operatorunun **elif** şaxələnməsindən istifadə etmək məsləhətdir.

Python dilində "qrup halında" mənimsətmə mümkündür. **uz_B = uz_k = 0** yazılışı düzgün olduğuna görə, öncə **uz_k** dəyişəninə **0** mənimsədilir, daha sonra **uz_k** dəyişəninin qiyməti **uz_B** dəyişəninə mənimsədilir.

```
# Böyük və kiçik hərflərin faizlə həcminin tapılması
#
setir=input("Mətni daxil edin: ")
uz_Setir=len(setir)
uz_B = uz_k = 0
for i in setir:
    if 'a'<=i<='z':
        uz_k+=1
    elif 'A'<=i<='Z':
        uz_B+=1
print("Kiçik hərflərin %%-i: %.2f" % (uz_k/uz_setir * 100))
print("Böyük hərflərin %%-i: %.2f" % (uz_B/uz_setir * 100))
```

Qoşa faiz işarəsi ekranda faiz işarəsinin bir dəfə çapını təmin edir. Tək faiz olsa, sətrin formatının başlanğıcı başa düşülə bilər.

Məsələ. Verilmiş sətirdə ən qısa sözü tapmalı. Sözlər bir-biri ilə probellə, durğu işarələri ilə ayrılırlar.

Məsələnin qoyuluşu: **setir** adlı sətrə ixtiyarı bir mətn daxil edilmişdir. Həmin sətirdəki ən kiçik uzunluğa malik olan sözütapıb çap etməli.

Növbəti daxil edilən sözün uzunluğu **uz** dəyişənində saxlanılacaq. Proqramın əvvəlində heç bir söz olmadığına görə bu dəyişənə sıfır qiyməti mənimsədiləcək.

Ən kiçik uzunluğa malik sözün uzunluğu **min_uz** dəyişənində yadda saxlanacaq. Başlanğıcda bu dəyişənə ən uzun sözün uzunluğunu yazmaq lazımdır. Ən uzun sözün uzunluğu sətrin uzunluğundan çox ola bilmədiyindən,başlanğıc maksimal qiymət kimi sətrin uzunluğu götürülə bilər. Deməli, başlanğıcda **min_uz** sətrin uzunluğuna bərabər olacaq.

for əmri vasitəsilə sətri simvol-simvol araşdıracağıq.
Əgər növbəti simvol hərfdirsə, **uz** dəyişəninin qiyməti bir vahid artırılacaq. Əks halda bu növbəti sözün sonu olduğu üçün növbəti sözün uzunluğu əvvəlki minimal sözün uzunluğu ilə müqayisə olunur. Əgər növbəti sözün uzunluğu əvvəlki kiçik sözün uzunluğundan kiçikdirsə və **uz**sıfıra bərabər deyilsə, onda **min_uz** dəyişəninin qiyməti **uz** dəyişəninin qiyməti ilə dəyişdirilir.
Burada **if** əmrinin **else** şaxəsində yeni sözün uzunluğunu hesablamaq üçün **uz** dəyişəninə sıfır mənimsədilir.

```
# Böyük və kiçik hərflərin faizlə həcminin tapılması
#
setir = input()
setir = input("Mətni daxil edin: ")
uz = 0
min_uz = len(setir)
for i in setir:
        if 'a'<=i<='z' or 'A'<=i<='Z':
                uz += 1
        else:
                if uz < min_uz and uz != 0:
                    min_uz = uz
                    uz = 0
print("Daxil edilmiş sətirdə ən qısa sözün uzunluğu-",min_uz)
```

Məsələ. Sözlərdən ibarət massiv verilmişdir. Python dilində verilmiş müəyyən uzunluqdan böyük olan sözlərdə son üç simvolu əvəz etməli.

Məsələnin qoyuluşu: Sözlərdən ibarət massiv verilmişdir.Həmin massivdəki müəyyən ölçüyə (məsələn, 5-dən böyük) malik sözlərdə sonuncu üç simvolu **$** simvolu ilə əvəz edin.

Əgər növbəti sözün uzunluğu müəyyən uzunluqdan böyükdürsə, onda sözün əvvəlindən sözün sonuna 3 simvol əskik olana qədər uzunluqda kəsiyi götürüb **$** simvolu ilə birləşdirmək lazımdır. Daha sonra massivdə köhnə sətri yeni sətirlə əvəz etməliyik.

sozler[i][0:-3] ifadəsi onu bildirir ki, öncə massivdən cari sətrin kəsiyi götürülür.

```
sozler = []
for i in range(10):

        sozler.append(input("Növbəti sözü daxil et: "))
i = 0
while i < len(sozler):
        if len(sozler[i]) > 5:
                sozler[i] = sozler[i][0:-3] + '$'
        i += 1
print(sozler)
```

Proqramın yerinə yetirilməsinin nəticəsi:

```
Növbəti sözü daxil et: albalı
Növbəti sözü daxil et: alma
Növbəti sözü daxil et: gavalı
Növbəti sözü daxil et: şaftalı
Növbəti sözü daxil et: naringi
Növbəti sözü daxil et: portağal
Növbəti sözü daxil et: heyva
Növbəti sözü daxil et: fındıq
Növbəti sözü daxil et: limon
Növbəti sözü daxil et: ananas
['alb$', 'alma', 'gav$', 'şaft$', 'nari$', 'porta$', 'heyva', 'fın$', 'limon', 'ana$']
```

Mənbələr

1. “ИнформатикА” Учебно-методический журнал для учителей информатики. Издательский дом Первое сентября. № 9 2014г.
2. А. Н. Чаплыгин. Учимся программировать вместе с Питоном. Учебник. — ревизия 226. — 135 с.
3. Вабищевич П. Н. Численные методы. Вычислительный практикум. – – 320 с.
4. Доусон М. Программируем на Python. – СПб.: Питер, 2014. – 416 с.
5. И. А. Хахаев. Практикум по алгоритмизации и программированию на Python. Учебник. — М.: Альт Линукс, 2010. — 126 с. — (Библиотека ALT Linux). — ISBN 978-5-905167-02-7.
6. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
7. Лутц М. Программирование на Python, том I, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 992 с.
8. Лутц М. Программирование на Python, том II, 4-е издание. – Пер. с англ.
9. Марк Саммерфилд. Программирование на Python 3. Подробное руководство. — Перевод с английского. — СПб.: Символ-Плюс, 2009. — 608 с —ISBN 978-5-93286-161-5
10. Пилгрим Марк. Погружение в Python 3 (Dive into Python 3 на русском)
11. Прохоренок Н.А. Python 3 и PyQt. Разработка приложений. – СПб.: БХВ-Петербург, 2012. – 704 с.
12. Прохоренок Н.А. Самое необходимое. — СПб.: БХВ-Петербург, 2011. — 416 с.
13. Сузи Р. А. Язык программирования Python: Учебное пособие. — М.: ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. — 328 с. — ISBN 5-9556-0058-2,ISBN 5-94774-442-2
14. http://asvetlov.blogspot.com/2010/05/blog-post.html
15. http://pep8.ru/doc/dive-into-python-3/1.html
16. http://pep8.ru/doc/pep8/
17. http://pep8.ru/doc/tutorial-3.1/
18. http://pep8.ru/doc/tutorial-3.1/6.html
19. http://pythlife.blogspot.com/2014/08/blog-post.html
20. http://pythontutor.ru/visualizer/
21. http://pythonworld.ru/samouchitel-python
22. http://torofimofu.fvds.ru/learnwithpython/ru2e/index.html
23. http://webquant.ru/posts/data-structures/
24. http://www.opennet.ru/docs/RUS/python/idle.html
25. http://younglinux.info/python/task/string
26. https://www.ibm.com/developerworks/ru/library/l-python_part_4/
27. https://www.python.org/
28. http://pythontutor.ru/visualizer/
29. http://pythontutor.ru/lessons/2d_arrays/
30. http://pythonworld.ru/
31. http://belgeler.istihza.com/py3/

Elektron formatda pulsuz yayılmaq üçündür.





Abdulla Qəhrəmanov

İlahə Cəfərova

Abdulla Qəhrəmanov 1978-ci ildə BDU-nun Tətbiqi riyaziyyat fakültəsini bitirmişdir. İT üzrə 42 illik iş təcrübəsinə malikdir. Dövlət Statistika Komitəsində, Maliyyə Nazirliyi, Az. DETK Bakı ETM-də proqramlaşdırma və tədris şöbələrinə rəhbərlik etmişdir. 220 və 23 №-li məktəblərdə, Xəzər Universitetində, "İstək" liseyində informatika və iqtisadiyyat fənlərindən dərs deyib. Kurikulum üzrə təlimçidir. Çoxlu sayda layihələrin iştirakçısıdır. 9 vəsaitin 200-dən artıq elektron resursun, 3 səmərələşdirici təklifin müəllifidir.

İlahə Cəfərova 2003-cü ildə ADNA-nın Cihazqayırma fakültəsini İnformasiya-ölçmə texnikası və texnologiyası ixtisası üzrə bitirmişdir. 2008-ci ildən Bakı ş. 23 №-li məktəbdə İnformatika fənnini tədris edir. "Elektron məktəb", "1 şagird, 1 kompüter" layihələri üzrə təlimçidir. Müxtəlif növ elektron lövhələrdə peşəkar səviyyədə işləyir. 5 vəsaitin, 30-dan çox elektron resursun müəllifidir.
Bir çox təhsil sərgilərinin iştirakçısıdır.